Evolutionary optimization of market-based control systems for resource allocation in compute farms

Neil Robinson
Internet Storage and Systems Laboratory
HP Laboratories Bristol
HPL-2002-284
October 23rd , 2002*

market-based control, trader agents, zip traders, auctions, genetic algorithms, BICAS

This thesis describes the development of a market-based control (MBC) system used to allocate and balance computational tasks across a minimal simulated Utility Data Centre (UDC). Firstly, a re-implementation of the original ZIP trading-agent was developed and tested in a variety of basic markets. The re-implementation faithfully replicated the market equilibration behavior of the original system, and a evolutionary algorithm was then employed to successfully fine-tune the performance of the system. The MBC UDC simulation is then presented as the first of its kind based on ZIP trading agents and also being fully distributed and autonomous in its operation. Experiments indicate a successful proof-of-concept and an efficient computational load-balancing performance under different scenarios. Use of an evolutionary algorithm is then made to both further improve the market equilibration performance of the ZIP traders operating within the MBC simulation and to evolve the marketplaces they operate within. Thus this thesis presents the first ever (preliminary) results from using artificial evolution to automatically design the auction mechanism for an MBC system.

* Internal Accession Date Only
Approved for External Publication

Evolutionary optimisation of market-based control systems for resource allocation in compute farms.

Master's Thesis[1]

Neil Robinson
COGS, University Of Sussex, U.K.
Hewlett-Packard Labs, Bristol, U.K.

[1]This thesis was submitted as partial fulfilment of the requirements of the Master of Science degree in Evolutionary and Adaptive Systems from the School of Cognitive and Computing Sciences within the University of Sussex at Brighton, U.K.

Acknowledgements

By far my biggest thanks go to Dave Cliff who has almost singlehandedly steered me through the project and with whom I have had many an informative and inspiring discussion. Thanks also to Peter Toft and Chris Thornton who were always there for further advice and comment.

Contents

Chapter 1

Introduction

The creation and use of Utility Data Centres (UDCs), where upwards of 10^4 computers are all networked together (either locally or across several sites) has opened new avenues for utility computing and in the execution of computationally demanding applications [1]. Given this potential, issues surrounding the design and management of UDCs have become increasingly active topics of research.

The sheer magnitude of UDC systems suggests that managing the load of hundreds or thousands of simultaneously running compute jobs is a vital task, necessary to ensure that the underlying compute resources are used efficiently. This task becomes inherently more complex when it is noted that, in most cases of interest, the compute jobs are heterogeneous in nature and are constantly fluctuating in their demand for compute resources. Further, these dynamic changes in the demand for compute resources are complemented by dynamic changes in the supply for compute resources as UDC system components such as processors, disk drives, or network links go down either due to failure or for scheduled servicing.

Thus UDC's present a resource allocation problem: that of balancing the supply and demand of scarce compute resources. The resolution of supply and demand for scarce resources is a matter well framed within the field of microeconomics. Taking inspiration from these economic metaphors, a relatively novel technique termed Market Based Control (MBC) is proposed [2] as an ideal candidate to address the UDC resource allocation challenge. However, almost all of the resource allocation problems managed by existing MBC systems have been deficient by either being centralised (relying on a central controller) or not fully autonomous (relying on human input to some degree). Clearly an autonomous and decentralised MBC is required for UDC's.

In order to provide this decentralisation and autonomy, it is necessary for all computing nodes in a UDC to be potential suppliers of compute resource to outside entities, and also for an 'internal market' to operate so that jobs can be reallocated within the UDC as necessary. Such buying and selling of compute resources requires 'trader' software agents be attached to resources and to jobs. Thus, this project reports on the development and use of a UDC MBC system based on Zero-Intelligence-Plus (ZIP) trading agents, already well-proven as minimal, autonomous bargaining agents, able to adapt rapidly and smoothly to changes in supply and demand. To further fine-tune the resource allocation

efficiency of MBC control system, a genetic algorithm (GA) is employed to evolve the adaptation parameters of the ZIP-trading agents and of the marketplace that the agent operate within.

To the best of my knowledge, this is the first time ever that a GA has been used to automatically design the auction market mechanism for any MBC system.

The remainder of this report is organised as follows:

Chapter 2 – Background: provides an overview of the relevant background material. It introduces the concept of UDC's and the field of market-based control. Following this, an illustrative sketch of how an MBC system might operate in a UDC is presented, making clear the need for trading agents to act as interfaces to the UDC and also to act as arbitrageurs within the UDC. It concludes with an overview of the ZIP traders of Cliff [3], recently demonstrated by IBM researchers to outperform human traders [4].

Chapter 3 – A Re-implementation of Previous ZIP-Trader Research: presents a detailed description of ZIP traders, and presents selected results from re-implementing the basic ZIP trader markets used in 1996/97 [3]. Following this, replication of more recent results where a GA is used to find appropriate settings of key ZIP-market initialisation parameters are also presented. The results presented in this chapter demonstrate that relevant prior work has been replicated successfully.

Chapter 4 – MBC of a Minimal UDC Model with ZIP-Traders in Distributed Markets: Shows, for the first time, the successful use of ZIP traders for MBC resource-allocation in a minimal model of a UDC. Presents experimental results illustrating proof of concept and explores the performance of the resource allocation given different task-shifting policies.

Chapter 5 – Evolutionary Optimisation of the MBC System: presents use of a GA to evolve ZIP-trading agent parameter sets and marketplaces within the MBC simulation. The results presented show that, by use of evolutionary techniques, the market equilibration performance of the MBC has been improved upon.

Chapter 6 – Future Work and Conclusions: Details possible avenues for taking the work further, presents a summary of the main results, and reiterates the main findings.

Appendix A – Background Economics: this thesis assumes that the reader is familiar with the fundamental microeconomics of supply and demand that underlies MBC; but for completeness Appendix A presents an introductory overview of the relevant economics literature.

Appendix B – Additional ZIP-Trading Agent Experimental Results: presents additional results from experiments with the re-implementation of the ZIP simulation.

Appendix C – Source Code: contains the commented simulation source code.

Chapter 2

Background

2.1 Utility Data Centres and their Control

Utility Data Centres (termed UDCs or 'compute farms') are warehouse-style facilities where very large numbers of computers are networked together, containing upwards of 10^4 processor units. They offer supercomputing-scale provisions for general utility computing and a range of other computationally intensive applications such as genome sequencing and graphics rendering.

As UDCs are large and complex systems, this indicates that the control of such systems is of primary concern in ensuring both a cost-effective resource utilisation and an optimal quality of service (e.g. resource availability) to the end user. In order to simplify the management of UDC and PSC systems, a *service-oriented* view is adopted where a number of abstractions and separations to the system structure are made [5]. For example, the overall 'control space' of the system is divided into a number of interacting *control layers*. The *resource control* level that manages the physical resources of the UDC represents the lower-level control layer – in summary the task of this layer is to optimise the placement of services on servers, monitor the balance, and then re-deploy whenever the balance is lost. It is this task of balancing the demanded and supplied capacity that is the main interest of this thesis. Further, the number of controlled objects in the system is reduced by making them of higher granularity so that they are both uniform and simple.

However, even given these abstracted simplifications, traditional methods of system management that often rely on a central controller process of some sort simply cannot meet the necessary scale and reactivity requirements for UDCs [5]. Thus, a number of different control algorithms have been proposed which aim to decentralise the decision making of service placement across a UDC, the overall idea being that these algorithms seamlessly fit within the overall control infrastructure of the UDC (part of the 'intelligent' middleware).

The algorithms have ranged from using Integer Optimisation techniques [5] which give a highly accurate but slow resource allocation, through to an Ant Colony Optimisation (ACO, [6]) approach which is faster but more approximate in its allocation. Further up this scale, the 'Agents in overlay networks' [6] and 'Broadcast of Local Eligibility' (BLE)-based algorithms [6] are, along with ACO, fundamentally agent-based approaches which have many advantages in enabling a distributed schema of control. The autonomy and local decision making made

possible by these approaches facilitate self-organisation, a fast reaction time and a scalability devoid from traditional control architectures. Nonetheless, taking into account the issues of global accuracy and complexity, they are not perfect and thus improved agent-based approaches are sought. The use of economically motivated agents within a MBC approach is one proposal.

2.2 Market-Based Control

The application of processes derived from free-market economics in adaptively controlling complex distributed systems has increasingly become an attractive proposition. In particular, the allocation of scarce resources by groups of self-optimising individuals is a central concept to microeconomics and is directly relevant to the allocation and adaptive control of scarce computational resources in a UDC. The application of economic principles in the allocation of scarce resources is termed *market-based control* (MBC). There are several conceived benefits to employing the MBC approach; primarily, the decentralised nature of such systems offer robustness and a graceful degradation to failed nodes and links that is unmatched by brittle, centralised controllers.

2.2.1 How MBC Works

Operationally, MBC systems consist of groups of software *trading agents* that interact within a pre-determined market structure and use a supplied currency to buy or sell the commodity that represents the scarce resource. The seller agents within the market are assigned to unit(s) of the commodity to sell, and the buyer agents purchase this commodity on behalf of an entity that requires some amount of commodity in order to perform some goal (e.g. in routing calls across a telecommunications network, individual calls may be the resource-requiring entities, and network bandwidth would be the required resource). Elementary economic principles regarding the global (market) supply and demand of the commodity in question determine the price. If there is a high demand for a commodity, its price will rise; and if demand lessens, its price will fall. It is usually desired that transaction prices will eventually converge to a stable *equilibrium price* as the true cost of the good is sought by the trading agents. The optimal balance of supply and demand is achieved at this equilibrium price, and so transactions at the equilibrium price represent an optimal and fair allocation

of resources (is *Pareto-efficient* [7]). Thus, in a similar vein to the other agent-based approaches mentioned in the previous section, the local decisions made by the trading agents in the system lead to a globally efficient allocation, or in computing terms, load balance.

2.2.2 Market Structure

Agents can interact within any of a number of different market mechanisms, and each has its own specific characteristics. Easily the most common market structure is the *auction*, a familiar example being the *ascending bid* or *English* auction, where, under co-ordination from an auctioneer, buyers make increasingly higher bids for an item until no one makes any further bids. The highest remaining bidder purchases the item. The *descending offer* or *Dutch* auction is in basic terms an opposite of the English auction where the sellers announce decreasing offer prices until a buyer accepts.

However, it is noted that in these auctions a central auctioneer is required and this goes against the decentralised principles of the MBC approach. The *continuous double auction* (CDA) offers an alternative market structure; here a group of buyers and sellers interact by announcing bids or offers at any time, often simultaneously and asynchronously transactions occur whenever a buyers bid and a sellers offer is accepted. Given the asynchronous and decentralised properties associated with the CDA it is perhaps no surprise that this mechanism is the most suited for use in an MBC system – indeed, much of the interest in the CDA stems from the fact the many of the world's financial markets have long been based around such a structure.

Within an MBC implementation, all the agents are privy to information regarding transactions that have occurred between other agents in the system, and possibly also to information regarding 'failed' bids and offers. This information consists of whether the last quote was a bid or an offer, the price of the bid or offer and whether a transaction took place. Thus an agent can make a competitive bid or offer based on the prices that are currently being asked for a commodity in the market and according to the prices at which transactions are occurring. This process plays a part in the system's overall approach towards the market equilibrium price for the commodity.

2.2.3 A Simple MBC System Example

In order to aid a greater understanding of the principles underlying MBC, a simple example of how a such a system could operate in a UDC is now described using basic notation.

Firstly, let the UDC 'network' consist of a set of servers, denoted by S_1, S_2, ..., S_n. A given server can talk to any other server on the network, and each has a capacity, denoted C_i, for 'resource units' – this could be processing power or disk storage. The compute tasks that come onto the UDC have a required resource capacity and a duration (for example, a required amount disk-storage for a certain duration of time).

For this example, let the UDC contain $n = 4$ servers, with the resource unit capacities C_1 to C_4 being 7, 4, 3, and 2 respectively. To begin with the UDC is populated with some initial loadings (denoted L_1, L_2, ..., L_n), and for the servers S_1, S_2, S_3 and S_4 these are 4, 3, 1 and 1. The respective loads and capacities on the network N can then be denoted as pairs (L_i, C_i) as follows: $N = ((4,7)(3,4)(1,3)(1,2))$.

Now, say a compute task comes in with a required resource unit capacity of 5. The place where the currently unassigned tasks are is denoted as the market M, and so the market now becomes $M = (5)$. Now the resource allocation procedure commences. Firstly, Server S_1 notes that it could run the new 5-unit task if it could buy resources elsewhere for its (currently running) 4-unit task, so it puts out a bid for a 4-unit task. As a result, the market now contains two tasks, $M = (4,5)$. Server S_2 then notes that it could run the 4-unit task if it could buy resources elsewhere for its 3-unit task, so it puts out a bid for a 3-unit task. The market becomes $M = (3,4,5)$. Lastly, Server S_3 notes that it could do the 3-unit task if it could buy resources to do its 1-unit task, so it puts out a bid for a 1-unit task. Now the market consists of 4 compute tasks (as identified by their respective resource unit capacity requirements), denoted as $M = (1,3,4,5)$.

Next, server S_4 takes the 1-unit task off the market, so now $N = ((4,7)(3,4)(0,3)(2,2))$ and $M = (3,4,5)$. This leaves Server S_3 free to take the 3-unit task, giving server loads and capacities of $N = ((4,7)(0,4)(3,3)(2,2))$ and the market as $M = (4,5)$. Server S_2 is then free to take the 4-unit task, leaving $N = ((0,7)(4,4)(3,3)(2,2))$ and $M = (5)$. And lastly, Server S_1 takes the 5-unit task with $N = ((5,7)(4,4)(3,3)(2,2))$ and $M = (.)$.

So in conclusion it is seen that the load has been dynamically re-distributed over the UDC network to accommodate the new task. Now if a new 1-unit or 2-unit task comes in, there is spare capacity on Server S_1. However, if the case arises that a task with a resource requirement of more than 2 units comes in, then either it would just have to wait, or alternatively a server with the *potential* capacity could try to unload its current commitments. Then if there are no takers the server has the option of dropping its current commitment and taking the new task if the new task will pay a higher price for the resource.

In the example above, the loads and capacities on the UDC network ended with $N = ((5,7)(4,4)(3,3)(2,2))$ and the market finished containing no tasks, $M = (.)$. But, say, if a new 3-unit task comes in, servers S_1, S_2 and S_3 could all potentially run it. If the new 3-unit task pays more than their current loadings then a little auction is run between the resource sellers (Servers S_1, S_2 and S_3) and the owners of the tasks. On the assumption that a higher purchase price is a fair reflection of greater need, then the tasks that get run are those that most need to be done. This means that the agents are re-negotiating prices dynamically – prices rise in times of high demand and fall in times of reduced demand.

However, as mentioned, this is a trivially simple example. The system is centralised in that every server can see the whole market. The UDC-based MBC system described in Chapter 4 is based on an arbitrary number of markets that are topologically distributed across the network of servers. Nonetheless, the example given provides an insight of the core supply and demand principles used by the MBC system to allocate and distribute resources.

2.2.4 Previous Work using MBC

Conveniently, much of the early MBC work is documented in a published collection by Clearwater [8] which contains applications of MBC in a number of diverse areas such as in the allocation of tradable pollution permits and in the provisioning of air in an air-conditioning system of a building. However, this work has been amply reviewed elsewhere [9], and given the reported deficiencies of being either not fully decentralised (relying on a central controller or process of some kind) or not fully autonomous (relying on human input to some degree), they are not considered further here. Instead, a couple of recent applications of MBC are reviewed.

Resource Allocation over the Computational Grid

Wolski, Plank and Bryan [10] have applied an MBC system to optimally allocate resources over the Computational Grid. This is in effect a 'power grid' in which applications running over the internet plug into in order to obtain computational resources to execute, in much the same way that appliances plug into the national grid for electricity. The rationale for using MBC within this domain is highlighted in [10]: 'modelling the Grid as a commodities market is natural since the Grid strives to allow applications to treat disparate resources as interchangeable commodities'.

Within this system, resource producers and resource consumers are clearly and separately defined – a *producer model* involves two different types of commodity (one being CPU and another disk storage) spread over a network representation of the Grid, and consumers express their needs to the market in the form of tasks. Each producer and consumer respectively uses a producer supply function and a consumer demand function in order to calculate how much to bid or offer for resources. The differences in the functions means that the producers will consider long-term profit and past performance when deciding to sell whereas consumers will spend opportunistically according to periodic budget replenishments. Both a commodities market (trading multiple commodities) and a second-price auction are employed for comparison in system experiments. Referring to theoretical work from Smale [11] regarding price adjustment towards equilibrium, it was concluded that, as Smale's results held for the simulated Grid, the use of a modified price adjustment mechanism within a commodities market approached the equilibrium price faster than the auction market used.

Because of this fairly theoretical analysis of price dynamics, it is difficult to tell how efficient this MBC implementation was in controlling resource allocation as no comparisons were made with more traditional methods. The consumer and producer efficiency statistics did however suggest that favourable results were achieved. Description of the system architecture was also sparse but it is assumed that the system described operates both autonomously and in a decentralised fashion (which, in the case of the commodities market, it does).

Routing in Telecommunications Networks

Gibney, Jennings, Vriend and Griffiths [12] discuss an MBC system to optimise call routing over a simulated telecommunications network. This problem is of crucial importance to a network provider whose profit may be compromised by lost calls brought about by inefficient routing over the finite bandwidth resources of the network.

In its entirety the system described consists of three layers; the bottom layer is the network over which calls are to be routed, the middle layer is the MBC system that performs the network management function, and the top layer is the user layer from which the calls originate. The MBC system is itself layered and contains multiple market types and agent types – this architecture is an artefact of the way the system operates. In summary, so-called link agents trade the underlying network resources (network link capacity) to sell on in link markets to path agents which buy bandwidth capacity across the network to form routes between a source node and a destination node. Lastly, call agents purchase a given path from path agents in order to route an incoming call.

Although at first this seems fairly complex, using adaptive pricing and inventory strategies the agents are able to route calls with reasonable success. However, the performance of the system only starts to show an improvement over conventional static routing when network call traffic is relatively dense and call duration is long. At the time of writing the system was still largely at a proof of concept stage and so many avenues regarding system improvement are proposed [12]. But, unlike a few of the earlier examples in MBC [8], the system is both fully decentralised (through use of the double auction market) and autonomous in its operation.

2.2.5 DAI and the Contract Net Protocol

Although not part of MBC, an important early predecessor to it was the field of Distributed AI (DAI) and the development of the Contract Net Protocol [13], where a number of analogies to and features of MBC can be traced back to. Specifically, the contract net protocol facilitates distributed problem solving by decomposing a (for example, resource allocation) task into a number of sub-tasks and sharing these among idle compute nodes through means of negotiation. This negotiation affects the task distribution among a network of

nodes (and thus the efficiency), but once distributed and in turn accomplished the tasks can solve a number of 'goals' such as distributing control and data to avoid bottlenecks, and allocating resources [13]. Back in 1980 when the contract net protocol was first developed, a number of advantages over traditional methods such as providing a means of distributed control and exhibiting graceful degradation meant that it was very much a forerunner to the MBC systems of today. However, the node-to-node negotiation of the task distribution (which marks out the difference between the control net protocol and MBC systems) has limitations in its rudimentary mechanisms and for making the actual task distribution decisions, and unlike MBC systems had no formal model for announcing, bidding and 'awarding' decisions.

2.2.6 MBC Summary

It is clear that MBC is a rapidly developing field and one that is being applied to a wide diversity of problems in distributed resource allocation and control. It is also apparent that there is no 'standard form' MBC system – the architecture of the two systems reviewed in Section 2.3.4 differ in almost every detail, although the underlying principles remain the same. Tailoring an MBC system to a particular control problem is evidently very important and the large number of parameters in such systems means that the fine tuning of the system can be crucial to ensuring a superior performance. In this respect the examples surveyed still require more development, but are able to demonstrate able use of MBC principles within each domain.

2.3 ZIP Trading Agents

2.3.1 Rationale

For the MBC system to be both fully decentralised and autonomous, it is required that the trading agents be able to act on bid or offer prices both intelligently and autonomously. Thus the bargaining mechanisms made possible by a basic machine learning rule are desirable. Zero-Intelligence-Plus (ZIP) trading agents were the first trading agent to be developed (by Cliff [3]) with such a 'minimal intelligence' capability, and in their basic form will be employed within the MBC system introduced in Chapter 4.

However, it has not always been obvious that any trading agent intelligence be required at all. Gode and Sunder [14] presented results using *zero-intelligence* (ZI) agents which suggested that by means of random stochastic bids and offers the market would still approach an equilibrium price in very similar circumstances to that achieved in earlier experiments with human subjects [15]. In light of this, Gode and Sunder claimed that the dynamics of market activity was largely if not solely due to the market structure in operation. But this was subsequently shown to be an inaccurate claim [3], as Gode and Sunder's results turned out to be an artefact of the implementation they used.

Thus it was proven that to obtain market behaviour (in CDA markets) similar to that obtained by human agents, some form of intelligent bargaining mechanism was necessary, and so the rationale for ZIP trading agents endowed with a minimal learning capability was established. More recently, the interest surrounding ZIP traders has increased following results obtained by Das et al. [4] showing that they can outperform human traders.

The specific details of how ZIP trading agents work is amply covered in Section 3.2 and so is not discussed further here. However, a brief review of later work on ZIP traders by Cliff [16] and Van Montfort [17] is now made.

2.3.2 Evolutionary Optimisation of ZIP-traders and Marketplaces

For the ZIP-trading agents to perform reasonably well (i.e. to be efficient allocators and to trade at the market equilibrium value) it is required that the parameters governing the adaptation of the agents be hand-tuned to specific values. To get around this trial and error means of setting the parameters, Cliff [16] used a genetic algorithm to automate the process. The results showed that the GA was able to evolve ZIP agent parameter sets that gave improved market equilibration performance over the manually picked values.

As is discussed by Cliff in [16], the automatic optimisation of ZIP-trading parameters will be crucial in more realistic and complex markets involving, for example, continuous markets and situations where agents receive noisy and delayed information (see below). Thus it can be concluded that use of evolutionary techniques will be invaluable in improving the performance of an MBC system used to control very large scale and high-complexity distributed systems such as a UDC. Finally, Cliff [18, 19] also looked at the evolutionary optimisation of

the market mechanisms in which the agents operate. The resulting hybrid-like market structures were shown to give even greater allocative efficiency.

As a final point regarding the delays and noise inherent from complex distributed systems, it is a requirement that the trading agents be robust in the face of limited knowledge. It is noted by Hogg and Huberman [20] that imperfect knowledge and delays in information may well introduce chaotic dynamics into a large autonomous computational system such as a UDC. The possible effects this would have on the resource allocation performance of an MBC system is a further consideration, especially concerning real-world deployment.

2.3.3 Arbitrageurs and Segmented Markets

Van Montfort [17] extended the use of ZIP agents to that of arbitrageur agents – a special kind of agent that can buy or sell commodities over any number of spatially distributed and partially overlapping markets. By doing this they can maximise their own profits by taking advantage of price differences in different markets, thus increasing supply in markets where prices are high and increasing demand in markets where prices are low. It was shown by Van Montfort [17] that arbitrageurs, acting only to increase their own wealth, can globally stabilise (i.e. equilibrate) spatially distributed and segmented markets. The use of such arbitrageur agents would be useful within a spatially distributed MBC system in order to obtain such price stabilisation dynamics across topologically local markets.

The simulation used by Van Montfort [17] does, however, have some limitations – for example, only buyer and seller agents *or* arbitrageur agents are permitted in any one particular market setup. As it would be quite fundamental to have all agent types operating both in local markets (buyer and seller agents) and globally across segmented markets (arbitrageurs), this would need to be resolved, along with other considerations such as the best neighbourhood structure to use between agents on the grid.

Chapter 3

A Re-implementation of Previous ZIP-Trader Research

3.1 Introduction

The Zero-Intelligence Plus (ZIP) trading agents were originally developed as minimal mechanisms able to engage in rudimentary bargaining behaviours in market-based environments. The original programs developed for the simulation were implemented in the ANSI C programming language and are shown as an Appendix in [3]. The first part of this chapter presents results using a re-implementation of the original ZIP simulation, in the Java programming language, as a prelude and basis to the MBC simulation introduced in Chapter 4. The second part of this chapter then presents a replication and extension of work in [16] using a GA to evolve ZIP agent parameter sets, in order to improve their market equilibration performance.

The program structure of the simulation remains to all intents and purposes the same as that in the original implementation with the exception of the necessary language porting changes. The aim of the re-implementation was primarily to gain a greater understanding of the simulation to which the MBC system would be based, and to ensure that it matched the results reported with the original implementation. The simulation source code for this re-implementation can be found in Appendix C.

3.2 Operation of ZIP-Trading Agent Simulation

This section will briefly summarise the main aspects concerning the design of the ZIP trading agents and the market structure of the simulation (for a comprehensive description of the original simulation, see [3]).

3.2.1 Initialisation and the Trading Procedure

Before the simulation is run, the supply and demand curves of the market must be specified. This is done by distributing a set of *limit prices* to all of the agents involved in the market. A buyer can't pay more than its limit price for a unit of commodity and a seller can't sell a unit of commodity for less than its limit price. A number of different supply and demand curves are used in the experiments reported in Section 3.3, as are markets (in Appendix B) where the supply and demand curves undergo a 'shock' change partway through the simulation.

Once running the simulation iterates over a number of 'trials'. In the spirit of the pioneering experimental economics work of Smith, each trial is run as a series of trading periods or 'days', within which each agent has the right to engage in a pre-set number of trades dependent on the number of commodity units the agent has the right to buy or sell.

At the beginning of each trading session an able agent is randomly selected to make a bid or an offer. If there are any willing agents (i.e. sellers if the quote was a bid or buyers if the quote was an offer) that are able to trade at this price then one of these is selected at random and the deal proceeds. The bank balances of the agents involved in the deal are updated and the trading strategies (profit margins) of all the agents are adjusted accordingly. This adaptation process is formalised below.

3.2.2 Adaptation of ZIP-Trading Agents

The rationale behind each ZIP trader agent is an ability to adapt (in order to maximise) a *profit margin* based on previous market activity, specifically the previous quotes and transactions made by other agents in the market. These quotes are identified by the quote price and whether the quote resulted in a successful trade or not, and determine *when* and by *how much* an agent should adjust its profit margin. It is this profit margin that the agent uses to calculate the price it quotes (offers or bids) in the market.

A number of factors verify the qualitative issue of *when* a profit margin should be adjusted [3]. Firstly, to do so the agent must be active in the market (i.e. still have an entitlement of units to buy or sell). After this, the agent refers to the last bid or offer made in the market and will alter its own profit margin according to the whether it was an offer or a bid, whether it resulted in a successful trade or not, and whether the last quote price, $q(t)$, was greater than or less than the price the ZIP trader would currently quote.

Briefly, a ZIP seller increases its profit margin whenever the last quote Q was accepted and its own quote price $p_i(t) \leq q(t)$. It reduces its margin only if it is still active and Q was an offer with $p_i(t) \geq q(t)$, or if Q was an accepted bid and $p_i(t) \geq q(t)$. Similarly, a ZIP buyer raises its profit margin whenever Q was accepted and $p_i(t) \geq q(t)$, and it lowers its margin when it is active and either Q was a rejected bid with $p_i(t) \leq q(t)$ or Q was an accepted offer with $p_i(t) \leq q(t)$.

The quantitative issue of by *how much* the profit margin $\mu_i(t)$ should be adjusted is controlled by a *Widrow-Hoff with momentum* machine-learning algorithm. Here the use of a learning rate parameter β and a 'momentum' parameter γ guide the current quote price of an agent towards some *target* price $\tau_i(t)$. In each ZIP trader i this target price constitutes a stochastic perturbation of the quote price $q(t)$, as follows:

$$\tau_i(t) = R_i(t)q(t) + A_i(t)$$

where R_i is a randomly generated coefficient that provides a *relative* perturbation of the last quote price $q(t)$, and $A_i(t)$, an additional randomly determined *absolute* perturbation. Using $U(c_l, c_u)$ to signify a uniform distribution over the range $[c_l, c_u]$, for buyers the R_i value is generated at random from $U(1.0 - c_r, 1.0)$, and for sellers from $U(1.0, 1.0 + c_r)$, whereas the A_i values for buyers are generated at random from $U(-c_a, 0.0)$ and for sellers from $U(0.0, c_a)$. The target price $\tau_i(t)$ is updated using the β_i and γ_i parameters by each trader i via the following rule:

$$\mu_i(t+1) = (p_i(t) + \Delta_i(t))/\lambda_{i,j} - 1$$

where $\Delta_i(t)$ is the Widrow-Hoff delta value, calculated using the trader's learning rate β_i:

$$\Delta_i(t) = \beta_i(\tau_i(t) - p_i(t))$$

Lastly, as mentioned, a ZIP trader computes its quote price $p_i(t)$ at time t for a unit j with limit price $\lambda_{i,j}$, using the trader's profit margin value $\mu_i(t)$. Specifically, this is calculated according to the following equation:

$$p_i(t) = \lambda_{i,j}(1 + \mu_i(t))$$

For all trading agents participating in the market, the learning rate β_i is initially assigned a value generated at random from $U(\beta_b, \beta_b + \beta_\Delta)$ where β_b equals 0.1 and β_Δ equals 0.4; the momentum parameter γ_i is initially assigned a value generated at random from $U(\gamma_b, \gamma_b + \gamma_\Delta)$, where γ_b equals 0.00 and γ_Δ equals 0.10; and the agents' profit margin μ_i is initially assigned a value $U(\mu_b, \mu_h)$ for sellers and $U(-\mu_b, -\mu_h)$ for buyers, where $\mu_h = \mu_b + \mu_\Delta$, μ_b equals 0.05, and μ_Δ equals 0.30. The c_r and c_a parameters are set to 0.05.

Finally, it is noted that all of these parameters are set identically to that in the original implementation [3] except the learning rate β_i which has been adjusted to match that used in [16] (in the evolution of ZIP parameter sets), and thus to be consistent with the results presented in later in this chapter.

This summarises how the agents calculate and adapt their quote prices in response to quotes made by other agents, be they successful or not. The result of this adaptive trading behaviour is a market that should quickly approach the equilibrium price as indicated by the intersect of the supply and demand curves. There are several means of measuring the activity of the market that makes analysis of this, along with other metrics, possible, and these are now outlined.

3.2.3 Measuring and Analysing Trading Activity

Several measures concerning the convergence of transaction prices towards the equilibrium price P_0 can be sought. The most prominent of these is that of Smith's α parameter, also termed the *coefficient of convergence*, for measuring the price stability for a good. It is computed at the end of each trading day within a trial, and is defined as $\alpha = 100\sigma_0 \ / \ P_0$, where σ_0 equals the standard deviation of transaction prices around the equilibrium price. Other metrics recorded at the end of each trading day in the simulation include that of *allocative efficiency*, which is a percentage expressing the total profit earned by all traders divided by the maximum total profit that could have been earned by all of the traders; *profit dispersal*, which is defined by [14] as the cross-sectional root mean squared difference between the actual profits and the equilibrium profits of the individual traders (the profit the trader would realise if all units are traded at the equilibrium price); and the mean transaction prices. Plots of these statistics will be used to illustrate the behaviour of the ZIP traders within the markets tested in the next section.

3.3 Experimental Results

A number of experiments with the re-implemented simulation were carried out with the aim of matching the results reported with the original C implementation [3], thus demonstrating that the replication was faithful. The purpose of the experiments were to not only test that the simulation would work as expected,

but to also explore market behaviour given various supply and demand schedules – i.e. different numbers of buyer and seller agents were tried with each set being assigned different limit price distributions. The ability of the ZIP traders to efficiently and profitably trade within changing market conditions is an essential attribute for use in a MBC system. The most prominent test results are presented in the Sections 3.3.1 and 3.3.2, with an analysis of the remaining tests given in Appendix B.

3.3.1 Symmetric Supply and Demand Curves

In the first test, a total of 22 ZIP-trader agents split into 11 buyer and 11 sellers were used with the limit prices for each uniquely chosen from the range of \$0.75 to \$3.25 in steps of \$0.25. This gives symmetric supply and demand curves, an equilibrium price of \$2.00 and an equilibrium quantity of 6 (Fig 3.1). Within this market, each agent could only buy or sell a single unit (i.e. only engage in one transaction). One simulation trial constituted a period of 10 trading days, and each trial was repeated 50 times and averaged so that all results were representative of actual market behaviour.

The trading statistic plots generated from this test are shown in Figures 3.2 through 3.7. The mean transaction prices plot (Fig. 3.2) clearly indicates a swift convergence of the market towards the equilibrium price of \$2.00, and this is confirmed in the plot of alpha values (Fig. 3.3), matching the results gained with the original implementation in [3]. The effectiveness of the ZIP agents in being able to maximise their profits is illustrated in the allocative efficiency plot (Fig. 3.4) by showing that after 4 trading days, the agents are able to earn the maximum total profit available. Further, the profit dispersion (Fig. 3.5) illustrates that the profit of the traders quickly equals that if all the units are traded at the equilibrium price. The quantity plot (Fig. 3.6) shows a stabilisation at the equilibrium quantity of 6 units, while the transaction-price time series (Fig. 3.7) from one trial also shows that from random initial profit margins the transactions prices stabilise at around \$2.00. The speed by which the ZIP agents are facilitating market equilibration is a positive sign and indicates that the simulation is working as expected.

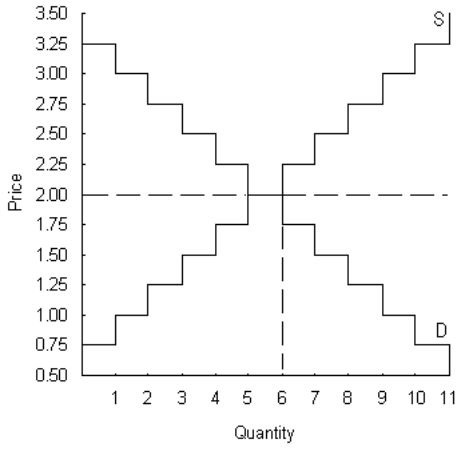

Figure 3.1: Symmetric supply and demand curves with 11 buyers and 11 sellers, where $P_0 = \$2.00$ (taken from [15] and [3]).

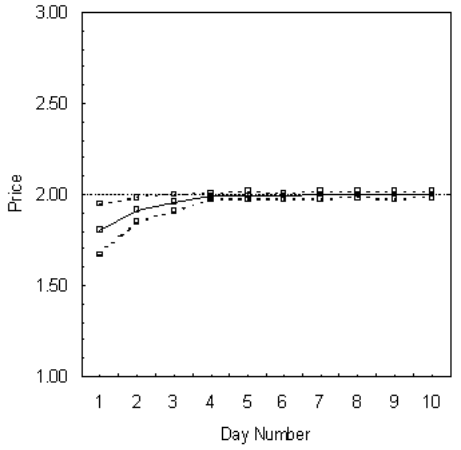

Figure 3.2: Mean transaction prices, averaged over 50 ZIP trials, for the supply and demand curves shown in Figure 3.1 ($P_0 = \$2.00$). The middle line is the mean value, upper and lower lines indicates the mean plus and minus one standard deviation.

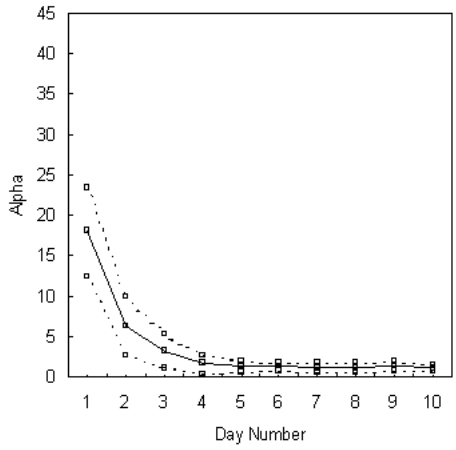

Figure 3.3: Smith's alpha value, averaged over 50 ZIP trials, for the supply and demand curves shown in Figure 3.1 (P_0 = \$2.00). Format as for Fig. 3.2.

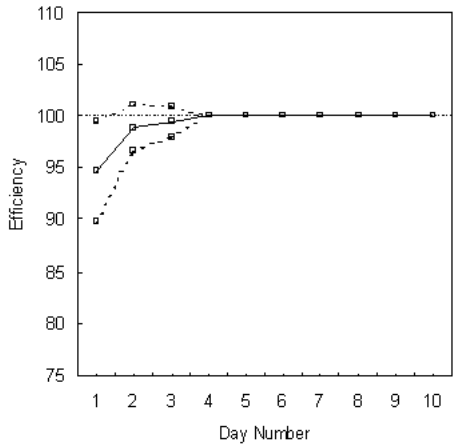

Figure 3.4: Mean allocative efficiency, averaged over 50 ZIP trials, for the supply and demand curves shown in Figure 3.1 (P_0 = \$2.00). Format as for Fig. 3.2.

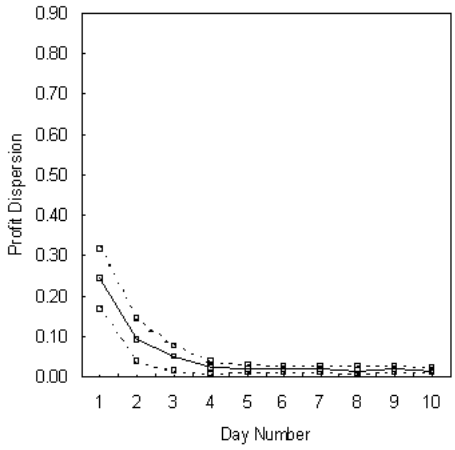

Figure 3.5: Mean profit dispersion, averaged over 50 ZIP trials, for the supply and demand curves shown in Figure 3.1 (P_0 = \$2.00). Format as for Fig. 3.2.

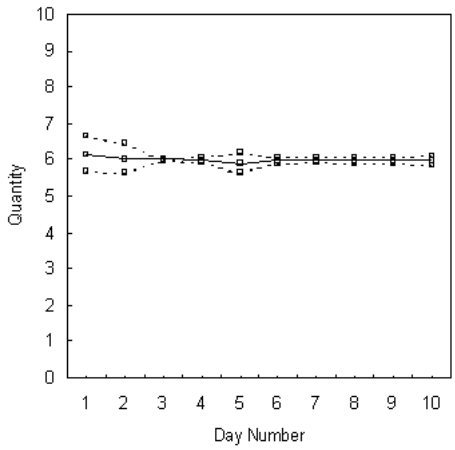

Figure 3.6: Mean quantity, averaged over 50 ZIP trials, for the supply and demand curves shown in Figure 3.1 (P_0 = \$2.00). Format as for Fig. 3.2.

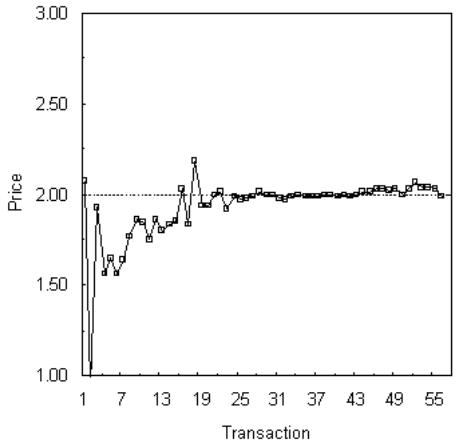

Figure 3.7: Transaction-price time series from one 10-day market ZIP trial using the supply and demand curves shown in Figure 3.1 (P_0 = \$2.00).

3.3.2 Flat Curves with Excess Demand

The purpose of this test is to clarify the market behaviour when there is a clear excess of demand. The flat supply and demand schedules shown in Figure 3.8 provide this, indicating that there are more buyer agents (11) than seller agents (6). Once again, the intersection of the curves gives an equilibrium price of \$2.00. The trading statistic plots generated from this test are shown in Figures 3.9 through 3.11, with a sample transaction price time-series taken from one of the individual trials in Figure 3.12.

The mean transaction price plot (Fig. 3.9) indicates a relatively slow approach towards the equilibrium price, although as with all of the statistic plots, this difference is purely an artefact of the particular ZIP agent parameter settings used (e.g. distributions of learning rate and momentum values) and could well be fine-tuned to improve the convergence [3]. Having said this, the overall convergence of transaction price and profit dispersion (Fig. 3.11) is an improvement to that shown in [3].

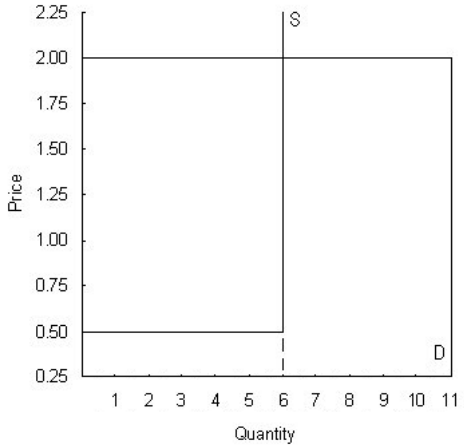

Figure 3.8: Flat supply and demand curves with excess demand: 11 buyers and 6 sellers, where $P_0 = \$2.00$

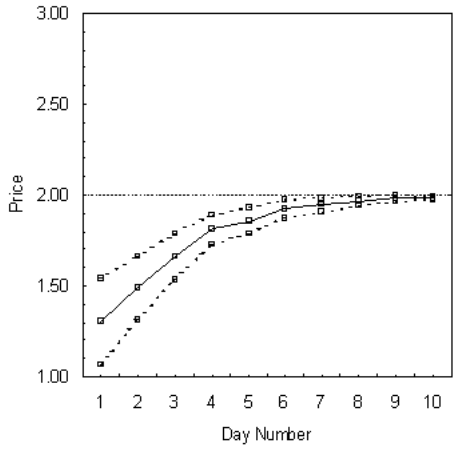

Figure 3.9: Mean transaction prices, averaged over 50 ZIP trials, for the flat supply and demand curves shown in Figure 3.8 ($P_0 = \$2.00$). Format as for Fig. 3.2.

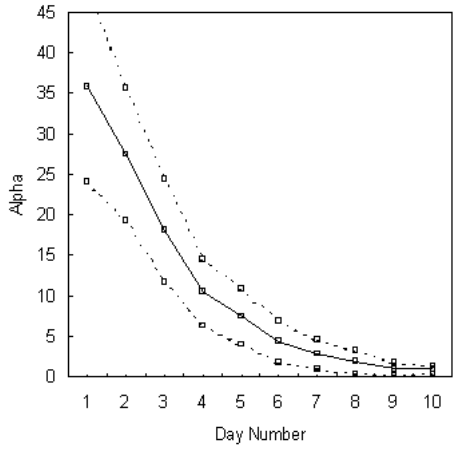

Figure 3.10: Smith's alpha value, averaged over 50 ZIP trials, for the flat supply and demand curves shown in Figure 3.8 ($P_0 = \$2.00$). Format as for Fig. 3.2.

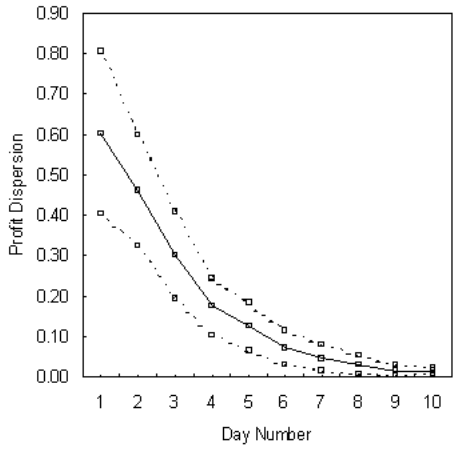

Figure 3.11: Mean profit dispersion, averaged over 50 ZIP trials, for the flat supply and demand curves shown in Figure 3.8 ($P_0 = \$2.00$). Format as for Fig. 3.2.

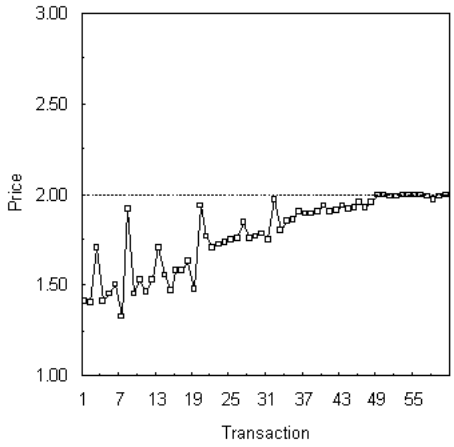

Figure 3.12: Transaction-price time series from one 10-day market trial using the flat supply and demand curves shown in Figure 3.8 (P_0 = \$2.00).

3.3.3 Summary of Results

This first set of experiments have indicated that the ZIP-trader re-implementation exhibits market equilibration in different market conditions (i.e. supply and demand schedules), and can adapt 'mid-trial' to changes in market conditions (Section B.2.3). The trading statistics plotted from each of the tests show that this is the case and closely matches the results gained with the original implementation. The next section looks at use of an evolutionary algorithm to optimise the ZIP agent adaptation parameters in order to fine-tune the performance of the system.

3.4 Evolutionary Optimisation of ZIP-Agent Parameter Sets

In all experiments carried out thus far, the values of the adaptation parameters used by the ZIP agents have been pre-determined – selected after a period of trial and error experimentation in order to obtain something in the range of the desired performance of the system. However, it was later shown in [16]

that by placing these parameters under evolutionary control, from a variety of initial parameter starting values, equally good and in some cases superior parameter values could be obtained that fine-tuned the trading performance of the ZIP agents involved in the market. The aim of this set of experiments is to replicate those carried out in [16] by using a simple genetic algorithm (GA) and a suitable evaluation function to evolve the adaptation parameters of the ZIP trading agents.

3.4.1 The Evolutionary Algorithm

A single evolutionary algorithm implementation was used in all the following experiments. A simple generational GA was employed using linear rank-based selection, a form of uniform crossover and elitism. The procedure was as follows:

```
For n times around the generation loop
   Evaluate fitness of all genomes in the population
   Select preferentially the fitter genomes as parents
   For <population size> times around the reproduction loop
      Pick 2 from the parental pool
      Recombine to make 1 offspring
      Mutate the offspring
   End reproduction loop
   Throw away parental generation and replace with offspring
End generation loop
```

Specifically, within the reproduction loop, a rank-based tournament selection was used where, for every member of the new population, three unique genomes were randomly selected from the existing (evaluated) population. The two fittest of these three genomes were then selected as the parents (denoted here by V_{mom} and V_{dad}) of a new offspring genome (V_{kid}). This method of selection was used to impose a fairly weak selection pressure, with the aim of preventing premature convergence to sub-optimal genomes.

Next, from the genomes V_{mom} and V_{dad}, the offspring genome V_{kid} was created, via crossover and mutation. As in [16], *stochastic multi-point crossover* was employed, whose first stage involves copying the first element of V_{mom} into the first element of V_{kid}. Then a uniform random value $x \in [0.0,1.0]$ is generated. If x is less than a threshold value T_x, then the copying process 'crossed-over'

so that the next element of V_{kid} would be copied from V_{dad}; otherwise copying continued from V_{mom}. This process of copying from the current parent and then crossing over to the other parent with probability T_x was repeated at each of the eight elements. Again, to faithfully replicate [16], T_x was set to a value of 0.125. Each element in the genome was then mutated by adding a real random value generated from U(-0.05,+0.05), clipping at 0.0 and 1.0 to ensure that $V_i \in [0,1]^8$.

Elitism was included to ensure that the fittest genome found so far was always retained. However, as pointed out in [16], because the overall ZIP trader market is stochastic, the fitness evaluation procedure is non-deterministic, and hence if the same population were to be evaluated twice, there is no guarantee that the same individual would be identified as the elite both times. The real-valued genome encoding of the agent adaptation parameters is now described.

3.4.2 Adaptation Parameters and the Encoding Scheme

As mentioned earlier, the value of the learning rate β_i, the momentum γ_i and the initial profit coefficient $\mu_i(0)$ parameters is assigned to each trader in the market using a random value from a uniform distribution [16]: thus, β_i is assigned a value from $U(\beta_b, \beta_b + \beta_\Delta)$; γ_i is assigned a value from $U(\gamma_b, \gamma_b + \gamma_\Delta)$; and the initial profit coefficient $\mu_i(0)$ is assigned a value from $U(\mu_b, \mu_h)$ for sellers, and $U(-\mu_b, -\mu_h)$ for buyers, where $\mu_h = \mu_b + \mu_\Delta$.

Therefore, the genome consists of the following eight real-valued parameters by which the adaptation of a ZIP-trader market is determined: the 3 pairs of bounds on the distribution of parameters for the individual agents ($\beta_b, \beta_\Delta, \gamma_b, \gamma_\Delta, \mu_b$, and μ_Δ), and the two parameters c_r and c_a (referring to section 3.2.2) that define the distributions of the stochastic perturbations used in calculating each agents target price. As a vector V, this is represented as follows:

$$V = [\beta_b, \beta_\Delta, \gamma_b, \gamma_\Delta, \mu_b, \mu_\Delta, c_r, c_a] \in \Re^8$$

A set of parameter values thus corresponds to a single point in the 8-dimensional space of possible parameter values.

In the experiments carried out earlier in this chapter, the default values of the these parameters were set as follows (to follow the convention used in [16], denoted as V_{cb}):

$$V_{cb} = [0.10,0.40,0.00,0.10,0.05,0.30,0.05,0.05]$$

To reiterate, there is nothing especially significant about these values other than that they were selected as a result of trial and error experimentation.

3.4.3 Fitness Evaluation Scheme

In order to facilitate a reliable comparison to the results originally presented in [16], the fitness evaluation of each genome V_i in the population has been exactly replicated. This was done by monitoring the price convergence (Smith's alpha) in a series of 50 independent trials, using the symmetric supply and demand schedules shown in Figure 3.1. At the start of each trial the parameter values represented on V_i were used to generate β_i, γ_i and $\mu_i(0)$ values for the 22 ZIP traders, as were the values of c_r and c_a used to calculate the traders' target prices in each experiment. As in [16], the experiment lasted for six trading periods. At the end of each day d, Smith's α measure was calculated (denoted $\alpha(d)$). The score for V_i on experiment number e, denoted by $S(V_i,e)$, was given by a weighted sum of the six $\alpha(d)$ values, where w_d denotes the weight on day d. In the trials, $w_1 = 1.75$, $w_2 = 1.5$, $w_3 = 1.25$, and w_4, w_5 and w_6 equal to 1.0. These weights place greater emphasis on the early trading days, when the ZIP traders are undergoing their initial adaptation to the market. The fitness evaluation function for V_i, denoted $F(V_i)$ was calculated as the arithmetic mean of $S(V_i,e)$ over $n = 50$ trials:

$$F(V_i) = \frac{1}{n}\sum_{e=1}^{n} S(V_i, e) = \frac{1}{n}\sum_{e=1}^{n}\frac{1}{6}\sum_{d=1}^{6} w_d \alpha(d)$$

In this scheme the GA is attempting to *minimise* the fitness score, with the optimum being $F(V_i) = 0.0$.

3.4.4 Experiments

Each experiment consisted of 50 independent evolutionary runs of 200 generations each, within which each genome, in a population of 30 genomes, was evaluated on the average of 50 trials.

Three sets of experiments were carried out, differing in the parameter values initially set on all of the genomes in the population – i.e. the choice of bounds on the distributions that generate the initial random population of V_i individuals. These determine the initial conditions of the evolutionary search, and have a significant effect on the success of the search. The results report on the best

evolutionary run from the 50 (i.e. the one giving the fittest elite genome at the final generation), for each of the three initial conditions. The three sets of initial conditions were as follows [16]:

- **Easy:** $V_i = V_{cb}$; $\forall i$
- **Zero:** V_i = [0,0,0,0,0,0,0,0]; $\forall i$
- **Hard:** $V_i \in [0.75, U_\Delta, 0.75, U_\Delta, 0.75, U_\Delta, U_c, U_c]$; $\forall i$ where $U_\Delta = U(0.00, 0.25)$ and $U_c(0.75, 1.00)$.

The purpose of starting an evolutionary search from the 'easy' initial conditions is to see whether and by how much the GA can improve (if at all) the fitness of the agents from the benchmark values used earlier in this chapter. The 'zero' initial conditions aim to illustrate whether the GA can find 'good' adaptation parameters starting from a highly sub-optimal parameter set (i.e. one that gives the agents no capacity for adaptation). Finally, the 'hard' initial conditions seek to obtain if the GA will again obtain fitter parameters and how these compare with the other two experiments should they be found. Given the results, the aim will also be to either confirm or question the original results presented in [16].

3.4.5 Results

The first set of plots (Figures 3.13, 3.14 and 3.15) show the fitness (weighted alpha value) of the elite genome at every generation for all 50 evolutionary runs, for each initial starting condition. For the 'easy' (Fig. 3.13) and 'zero' (Fig. 3.14) initial conditions, all 50 runs exhibit a (relatively) similar fitness level over all generations, the actual values of which can be considered even more similar given the stochastic nature of the ZIP fitness evaluation. Further, on the 'zero' plot it can be seen that the fitness values rapidly improve (after around 2 generations) from an expectedly very sub-optimal fitness score. For the 50 runs based on the 'hard' initial conditions (Fig. 3.15), there is clearly a much more noticeable divergence in the fitness of the elite genomes – a small group of evolutionary runs manages to maintain an increased fitness level that is very noticeable after 30 generations, then goes on to steadily maintain this fitness advantage through to the final generation. The group of runs above this are

more spread in their fitness levels throughout the 200 generations, indicating convergence to local optimums.

The best evolutionary runs from the 50 independent runs from each initial starting condition are now singled out for further analysis. To show a comparison of these 'best' runs, the fitness of the elite genome at every generation is shown in Figure 3.16. This plot shows the fitness level of the best 'zero' evolutionary run quickly converging to the same fitness level as the elite genome of the best 'easy' run, while in the best 'hard' evolutionary run, where initially starting with a better fitness than the 'zero' run, takes as long as 110 generations to attain the same fitness levels. By the final generation, all runs exhibit fitness levels that are a slight improvement over the initial 'easy' (V_{cb}) fitness score.

The following section presents an analysis of the best evolutionary run from the 'easy' initial starting conditions, while similar analysis from the 'zero' and 'hard' starting conditions can be found in Appendix B.

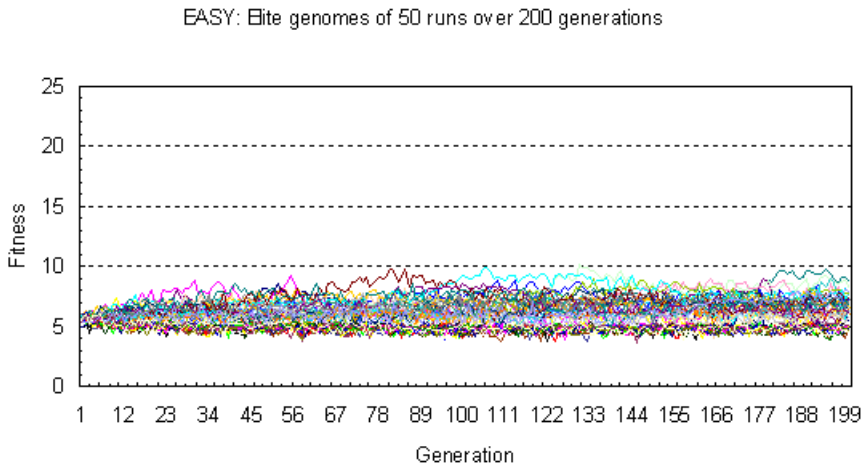

Figure 3.13: Elite genomes of 50 runs over 200 generations for the 'easy' initial conditions.

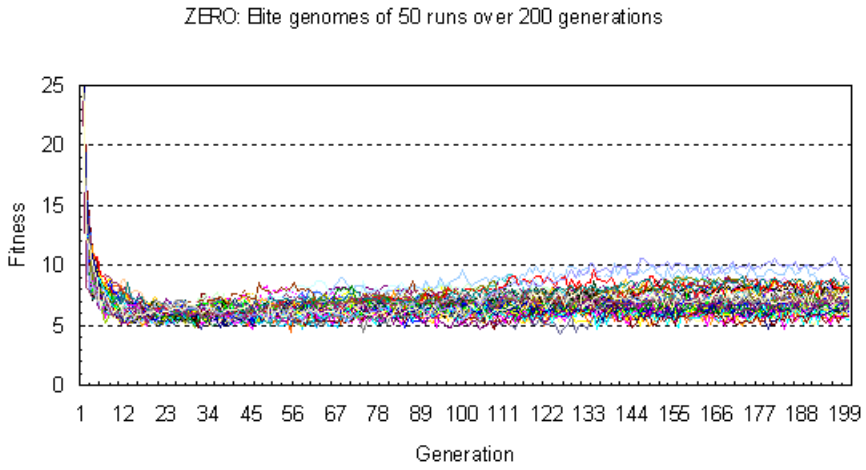

Figure 3.14: Elite genomes of 50 runs over 200 generations for the 'zero' initial conditions.

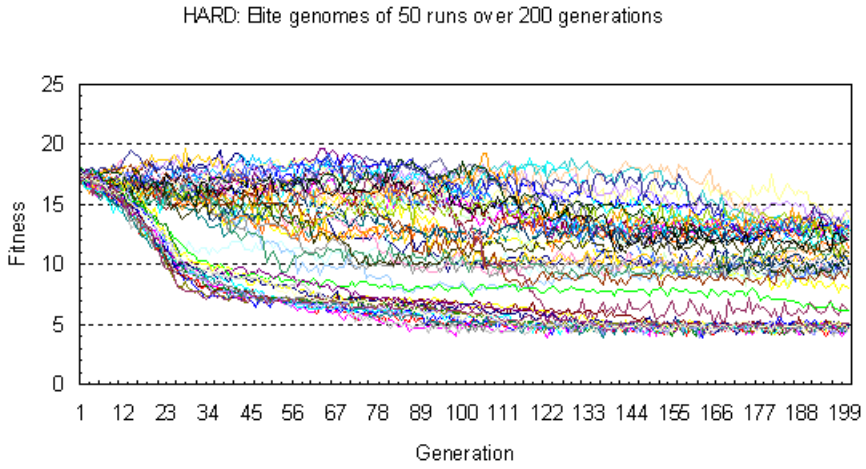

Figure 3.15: Elite genomes of 50 runs over 200 generations for the 'hard' initial conditions.

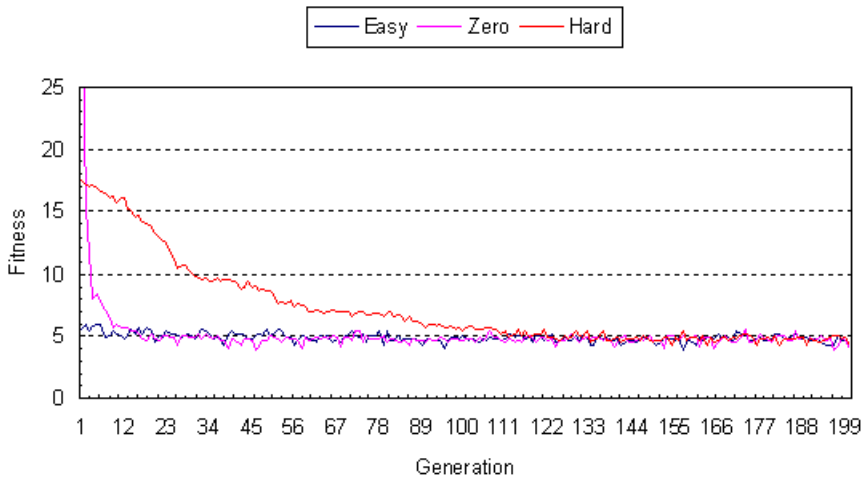

Figure 3.16: Fitness of the elite genome at each generation for each of the 'best' three evolutionary runs (Easy, Zero and Hard).

3.4.6 Easy Initial Conditions

To firstly clarify the behaviour of the market after zero generations, 6 trading days and using the default parameters V_{cb}, Figure 3.17 shows a sample transaction-price time series from one trial in the market of Fig. 3.1. To confirm the results reported in Section 3.3.1, it can be seen that after some initial volatility the transaction prices have closely converged to the equilibrium price of \$2.00. This convergence is confirmed in Figure 3.18, which shows an average of the α measure over 50 trials swiftly falling. By the 200th generation the transaction price time series (Fig. 3.19) has, as expected, converged more rapidly to the equilibrium price with subsequent quote prices diverging less from this equilibrium value. This is confirmed in the alpha plot (Fig. 3.20) which converges more closely to 0. The final generation elite genome for this run has the following parameter vector V_{easy}:

$$V_{easy} = [0.549, 0.041, 0.090, 0.178, 0.146, 0.000, 0.001, 0.061]$$

Apart from the β_Δ (2nd) parameter, the values on the V_{easy} vector that have resulted from this evolutionary run have matched surprisingly closely the

parameter values evolved in the original experiment [16], with six of the parameters having less than a difference of 0.1. In real terms, the V_{easy} parameter set favours a relatively high learning rate β_i, but a low momentum parameter γ_i and initial profit coefficient $\mu_i(0)$, and a very low perturbation (c_r,c_a) to the agents target price. The final fitness score of 4.45 also represents (as with the original results [16]), an improvement of the 5.49 weighted alpha score with the original V_{cb} parameter values, thus confirming that the evolutionary algorithm has successfully managed to fine-tune the market equilibration behaviour of the ZIP agents.

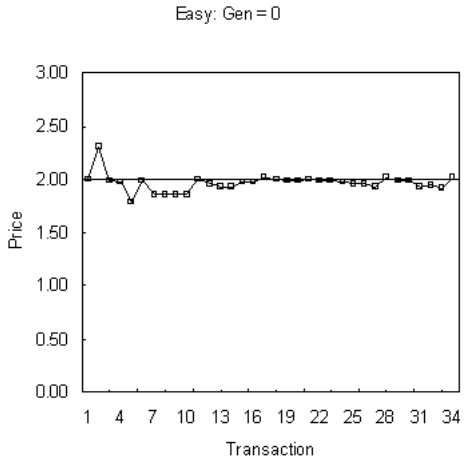

Figure 3.17: Transaction-price time series from one trial of the elite genome in the first generation from 'easy' initial conditions, with parameters set by V_{cb} (P_0 = \$2.00).

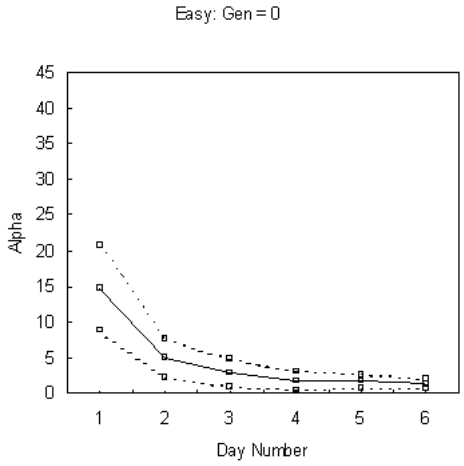

Figure 3.18: Smith's alpha value, averaged over 50 ZIP trials, for the elite genome in the first generation from 'easy' initial conditions, with parameters set by V_{cb} ($P_0 = \$2.00$). Format as for Fig. 3.2.

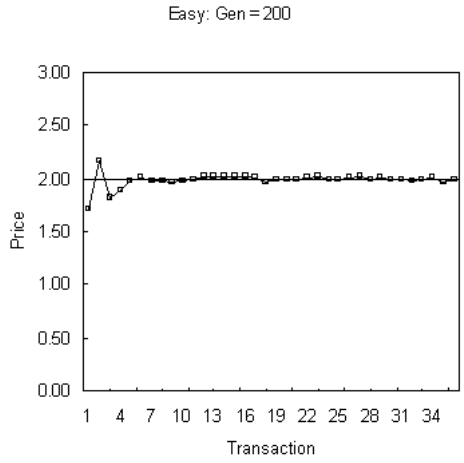

Figure 3.19: Transaction-price time series from one trial of the elite genome in the final generation from 'easy' initial conditions.

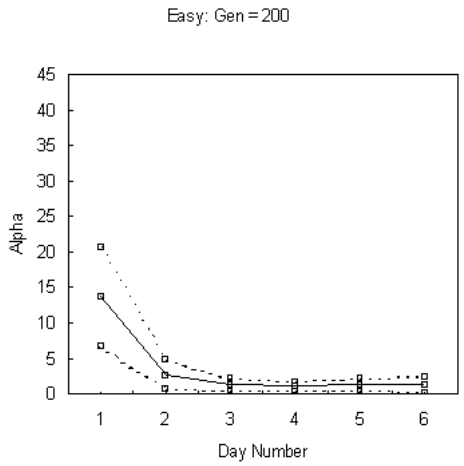

Figure 3.20: Smith's alpha value, averaged over 50 ZIP trials, for the elite genome in the final generation from 'easy' initial conditions.

3.4.7 Use of the Flat Curves with Excess Demand Market

In the results just presented, even though an improvement was made over the V_{cb} parameter values, the improvement was not huge, and as it is now proven that the V_{cb} parameter values are well suited to the supply and demand curves used (Fig. 3.1, see Section 3.3.1), they compare very well to the evolved values. This is where a further test (extending that originally presented in [16]) comes in – for this experiment the GA will evolve parameter values to the excess demand market of Fig. 3.8 – a market where it was shown in Section 3.3.2 that the original parameter values V_{cb} were not particularly well suited. Thus, the sole aim of this test is to identify whether the GA can evolve parameter sets in a market where the V_{cb} parameters did not originally perform well (i.e. that shown in Fig. 3.8). For this reason, only a single set of (25) evolutionary runs is carried out from the 'easy' initial conditions.

Fig. 3.21 shows the fitness of the elite genome at every generation for each of the 25 evolutionary runs. It indicates that after 40 or so generations, the GA has managed to find parameter sets that are very well suited to the Fig. 3.8 market. The 'best' evolutionary run from the 25 (Fig. 3.22) is now analysed.

The sample transaction price time-series (Fig. 3.23) and alpha plot (Fig.

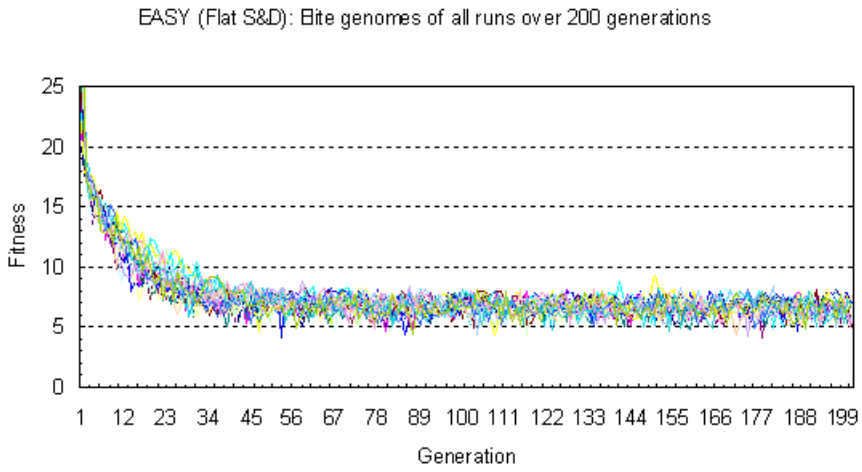

Figure 3.21: Elite genomes of 25 runs over 200 generations for the 'easy' initial conditions, using the flat supply and demand curves with excess demand as shown in Figure 3.8.

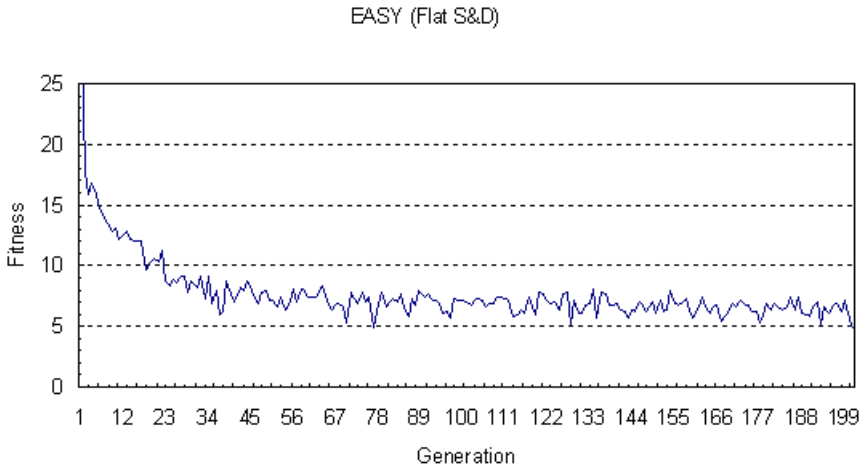

Figure 3.22: Fitness of the elite genome at each generation for the 'best' evolutionary run using the flat supply and demand curves with excess demand as shown in Figure 3.8.

3.24) confirm the results of Section 3.3.2 showing a slow approach towards the equilibrium price of $2.00. It was also noted in Section 3.3.2 that this convergence could be improved on by fine-tuning the agent parameter settings. Well, the sample transaction price time-series (Fig. 3.25) and alpha plots (Fig. 3.26) generated from the agent parameters of the final generation show very clearly that this has been the case. On closer inspection of the alpha plot, although the alpha value is fairly large in the early trading days, it swiftly converges to an extremely low value by the 4th trading period. The final generation elite genome for this run has the following parameter vector, denoted V_{flat}:

$$V_{flat} = [0.797, 0.167, 0.633, 0.095, 0.039, 0.067, 0.000, 0.028]$$

In comparison to the other evolved parameter sets (V_{easy}, V_{zero} and V_{hard}), all of the V_{flat} parameters are quantitatively similar (with the last three μ_Δ, c_r and c_a for all evolved vectors < 0.1), except the γ_b (third) parameter, where for V_{flat} it is noticeably larger in value. This parameter represents the lower bound by which the momentum parameter γ_i is calculated, so it can be concluded that for a market where there is excess demand (and flat supply and demand curves in this case) a larger momentum parameter is favoured, indicating a pronounced adaptation by the ZIP agents to quotes made in the market.

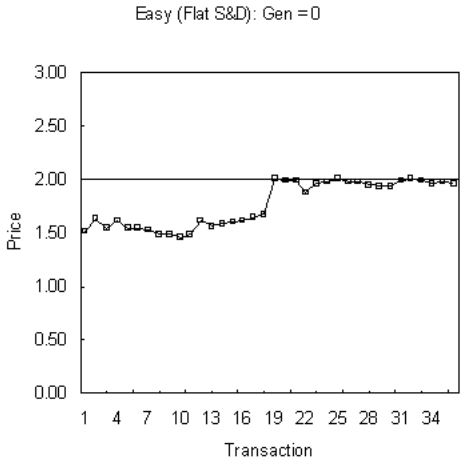

Figure 3.23: Transaction-price time series from one trial of the elite genome in the first generation from 'easy' initial conditions, using the flat supply and demand curves with excess demand as shown in Figure 3.8.

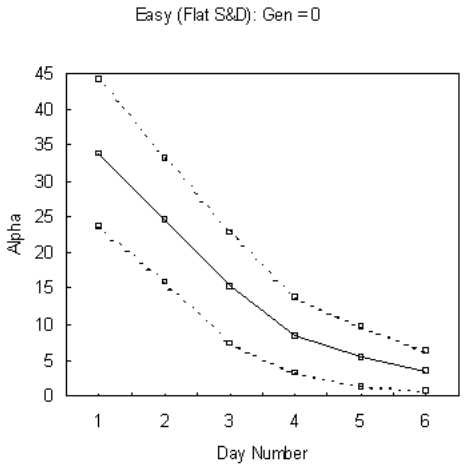

Figure 3.24: Smith's alpha value, averaged over 50 ZIP trials, for the elite genome in the first generation, using the market shown in Figure 3.8.

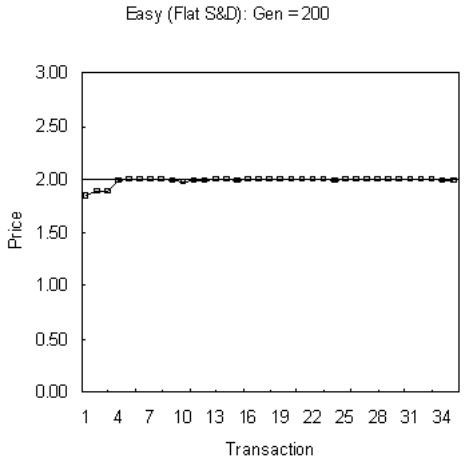

Figure 3.25: Transaction-price time series from one trial of the elite genome in the final generation from 'easy' initial conditions, using the flat supply and demand curves with excess demand as shown in Figure 3.8.

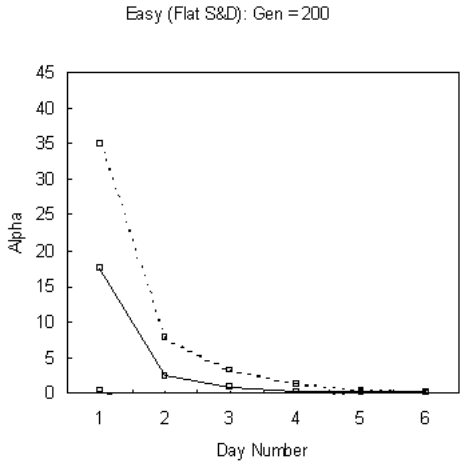

Figure 3.26: Smith's alpha value, averaged over 50 ZIP trials, for the elite genome in the final generation from 'easy' initial conditions, using the flat supply and demand curves with excess demand as shown in Figure 3.8.

3.4.8 Summary of Results

The results presented in this section have indicated that the ZIP agent adaptation parameters can be successfully fine-tuned. This is very beneficial given that manually-tuned parameters (such as those in the vector V_{cb}) cannot be exactly determined *a priori* to give the best possible market performance. The GA was also able to evolve agent parameter sets for a market where the initial parameter settings were not well suited.

The attention of this report now switches to that of extending the ZIP simulation for use within a distributed MBC system which is described further in the next chapter.

Chapter 4

MBC of a Minimal UDC Model with ZIP-Traders in Distributed Markets

4.1 Introduction and Rationale

In the previous chapter I demonstrated a replication of the original ZIP-trader system and also a replication of the use of a GA for optimising ZIP-trader market parameters. This demonstrated that the performance of the traders in maximising profit margins can be fine-tuned, further improving the equilibration of the market and the ability of the market to be 'well-damped' to shock changes in the market supply and demand curves. This chapter describes a novel adaptation of the ZIP re-implementation, into a market-based control (MBC) system used to allocate and balance compute tasks to resources over a simulated Utility Data Centre (UDC).

To model how ZIP trading agents could be used within such an MBC application domain, a distributed ZIP-trading agent system running over a minimal simulation of a UDC has been developed. Briefly, the interactions of the ZIP-agents acting on behalf of UDC servers and computational tasks occur in topologically local markets, trading compute resources in order to distribute the load appropriately. The following sections describe the operation of the system in more detail and experiment with a highly simplified example for proof of concept.

4.2 The MBC Simulation

4.2.1 System Structure

As this is the first study of using ZIP traders within a MBC system in UDCs, I chose to work with a minimal abstract simulation of a UDC. Use of a minimal simulation allows for the fundamental actions and effects of the ZIP-based MBC to be clearly identified and explored.

The UDC model itself consists of an arbitrary network of interconnected servers, each holding a number of resource units available to run computational tasks. Each server is assigned a buyer and a seller ZIP agent, and each computational task a buyer ZIP agent, whose logic and means of quote price adaptation remain as in the original ZIP re-implementation (Section 3.2.2). However, the logic concerning the initiation and execution of auctions is now held locally on every server. Significantly, this means that trades occur with topologically neighbouring servers in a distributed fashion, in effect spreading and balancing the computational tasks, enabling the MBC system to potentially scale to any

size of UDC, and also to exhibit graceful degradation given individual server failures. Each server has a local market to which new computational tasks 'arrive' in order to be first allocated resources on the UDC.

The number of servers and their connectivity are parameters of the UDC network. Thus any arbitrary server connectivity can be employed or specific neighbourhood function used.

4.2.2 System Operation

The MBC simulation runs for an arbitrary number of timesteps. At each timestep, the system randomly cycles through all servers in the UDC network until each has completed a single 'operation'. This server operation may be, for example, putting a task waiting in a local market up for auction or indicating if it is willing to participate in a neighbouring servers' auction. Specifically, each server is always in any one of the following five states: *Free*; *Pending Acceptance*; *Waiting Acceptance*; and *Waiting Market*. A given servers' state at timestep t+1 is dictated by the state of itself and of its n neighbouring servers at timestep t. The server states and the transitions between them is illustrated in Fig. 4.1.

4.2.3 The Resource Allocation Process

In order to best explain how the MBC simulation initiates and performs localised auctions for compute resources, a simple example scenario is now described in detail over several timesteps (for a simplified example see Section 2.3.3). 30 servers are used in this example, which can be thought of as being contiguously spread in a line. The connectivity of a given server is defined as the 5 neighbouring servers on each side (see Fig 4.2), so that each server has a total of 10 neighbouring servers. Servers at the end of the contiguous line 'wrap around' to the start of the line (is circular) in order to prevent any boundary effects from occurring. At the beginning of the simulation, all of the servers are running no tasks so that they have all of their resource unit capacity available. So in a case where only a single compute task enters the local market of Server 15, the MBC simulation proceeds at each timestep as follows (visualised in Fig. 4.3):

- **Timestep 1**: All servers on the UDC are in a *Free* state. When the MBC system (randomly) passes over Server 15, a single new task enters its local market. Server 15 detects the new task and (as it is in a *Free* state) puts

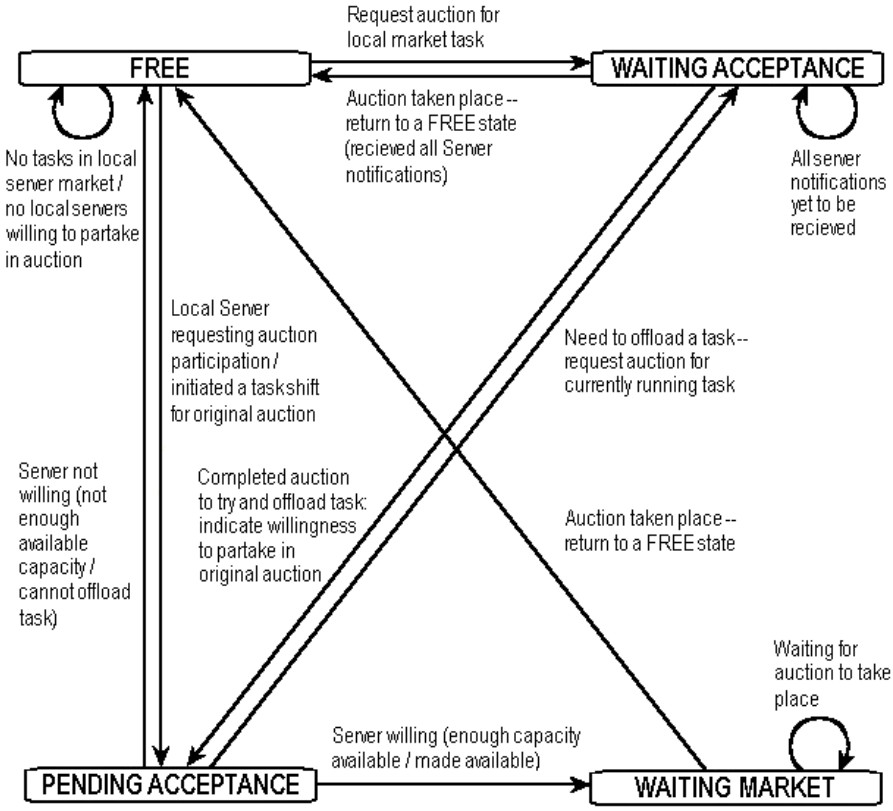

Figure 4.1: Server State Transition Diagram.

the task up for auction. As its neighbouring servers are also in a *Free* state, Server 15 changes the state of these neighbours to a *Pending Acceptance* state and sends it the bid price and resource unit requirements of the task. The state of Server 15 now becomes that of *Waiting Acceptance*, as it is waiting for notification of all neighbouring servers ability to partake in the auction. In the case where no neighbouring servers are in a *Free* state, in other words *not* able to participate in a new (prospective) auction, then this auction is terminated and the task returns to its local market for a potential re-auction in a future timestep.

- **Timestep 2**: The neighbouring servers, now in a state of *Pending Acceptance*, will indicate that one of their neighbouring servers (Server 15 in this example) has put out a bid price and a request for participation

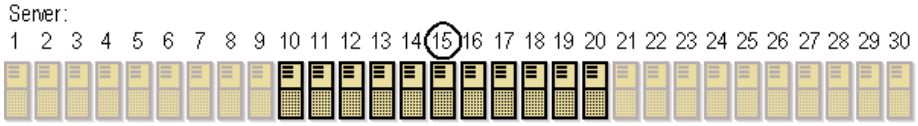

Figure 4.2: Local neighbourhood of Server 15.

in a local auction. As they are randomly passed over, each will return an indication of whether it is willing to partake in the auction. This is determined by taking the load requirements of the bidding task and verifying if the server currently has enough *available* capacity to potentially run the task. If this is the case, the server will change its state to that of *Waiting Market* to indicate its participation. If there is not enough available capacity but there is enough *total* capacity on the server (i.e. capacity subtracting that used by currently running tasks), one or more of these currently running tasks on the server may be put up for auction ('offloaded'), in order that the server can partake in the auction for the original task. However, whether this 'sub' auction is initiated within the process of the original auction depends on the state of the current task shifting policy in place (see Section 4.2.4). In the case where the server does not have enough total capacity to run the bidding task, the neighbouring server will indicate its exclusion from the impending auction by returning to a *Free* state.

- **Timestep 3**: When the system again passes over Server 15, it will check whether any of its neighbouring servers have indicated if they are willing to participate in the auction or not. This process can take several timesteps (i.e. the auction is put on hold) if one or more neighbouring servers attempt to offload any currently running tasks. When all neighbouring servers have indicated if they are able to participate or not, the auction proceeds (in the same timestep) as it would do in the original simulation with servers willing to sell resources according to the bid price put forward by or on behalf of the task. If successful, the task then 'moves' to the server who was successful in selling resources to it (see section 4.2.4 for how this is resolved). With the auction completed, Server 15 and all neighbouring servers go to a *Free* state until a new task enters a local server market.

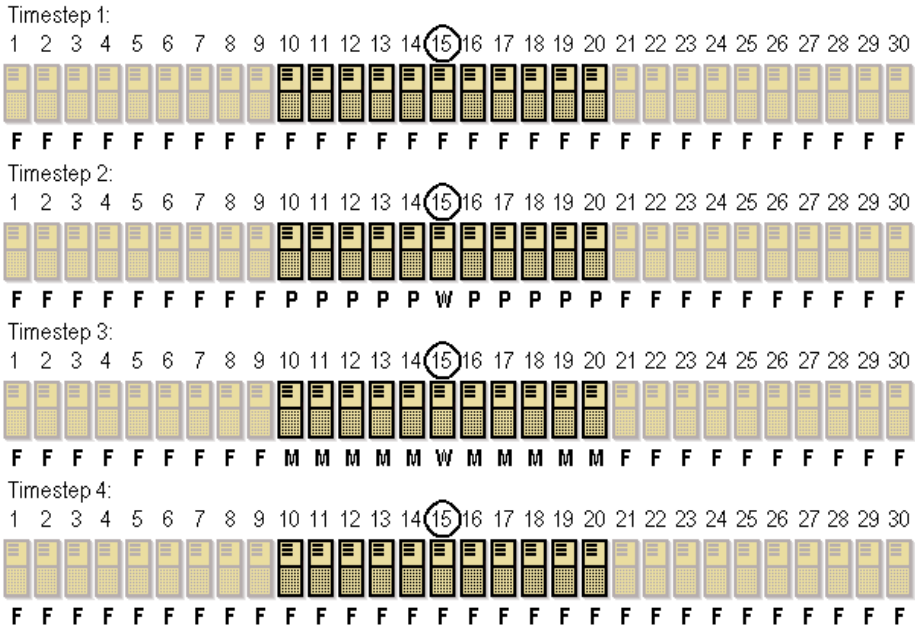

Figure 4.3: Server states at each timestep for the example scenario described in Section 4.2.3 (where F = *Free*, P = *Pending Acceptance*, W = *Waiting Acceptance* and M = *Waiting Market*).

4.2.4 Additional Extensions

There are a number of additional considerations that have to be taken into account regarding the operation and behaviour of the MBC system – for example, how the ZIP agent limit prices are set in order to determine the supply and demand curves in the global market, and how servers decide whether to shift currently running tasks in order to participate in auctions for new tasks. These are addressed in the following sections.

Setting of Agent Limit Prices

In the original ZIP simulation, the distribution of limit prices to buyer and seller agents corresponded to particular supply and demand curves. For example, 11 buyer and 11 seller agent limit prices set within a range of \$0.75 to \$3.25 in steps of \$0.25 gave an equilibrium price of \$2.00 (Fig. 3.1). But in the MBC system, larger numbers of servers and arbitrary numbers of tasks mean that the market

supply and market demand curve cannot be so precisely defined beforehand. However, to maintain similar 'market conditions' to those given by Fig. 3.1, all agent limit prices are still set within approximate ranges (for example between \$0.75 and \$3.25), even though the means by which this is done varies for each type of agent. The idea is to set limit prices sensibly so as to have buyer and seller agents' supply and demand curves crossing (otherwise no trades would occur) and have a realistic chance of successful trades occurring. Specifically:

- **Server seller agents** have their limit prices set in proportion to the total or available (dependent on the server auction eligibility criteria) number of resource units on the server. This means that a server with a proportionally higher resource unit capacity (as compared to all other servers on the UDC network) will have a proportionally higher limit price. This is justified as, in general, more resources cost more money.

- **Server buyer agents** have their limit prices set equal to whatever price the local server originally paid for the computational task (that it is trying to buy resources for), so that the server does not end up making a net loss on the task, should it offload. For this reason a small profit margin may also be added onto the limit price.

- Lastly, the limit price for a **task buyer agent** is set equal to the amount of funds the task has for buying resources, as intuitively the task buyer cannot spend more for resources than it has in funds to pay for it.

It is also noted that the task buyer agent will only be involved in an auction for the initial resource purchase and that after this any subsequent auctions involving the task will be dealt with by the server agents only as they will subsequently 'own' the task.

Task-Shifting Policies

As was described in the example given in Section 4.2.3, a situation will readily occur whereby a given server can participate in an auction for a task that has made a bid from a neighbouring servers' local market *only* if it attempts to offload a currently running task, in order to free up enough resource unit capacity. A decision must be made to either attempt to offload the currently running task (i.e. initiate a new auction), or to continue running the task and

reject participation in the neighbouring servers' proposed auction. This decision will be made against the auction rules used and thus is not a choice for the server to make itself. There are several different 'policies' that can be employed here, including 'always task shift'; 'task shift if the new task would pay more for resources than the task currently running'; and 'randomly shift one task'. Each has a significant effect on how tasks 'spread' and organise over the UDC network within periods of high task traffic, and are explored in the experiments carried out in Section 4.3. Also, it is noted that each market auction involves only a single computational task, in order that a (neighbouring) server can decide whether to unload a currently running task based on the resource requirements and priorities of the auction task only (i.e. the UDC network locally 're-organises' around the new task).

Closing Auction Deals

Within the process of an auction itself, if there is more than one willing server wanting to participate in a deal for a given task, then there are two means by which a task buyer or a server buyer agent can select a willing server (seller) to trade with and purchase resource unit capacity to run the task. In a *non-competitive outcome*, this server may be selected on a random basis with no need to maximise the profits of the individual servers or tasks involved in the deal. This is analogous to an intra-company situation where a UDC is utilised by a single organisation. On the other hand, use of a UDC may be loaned or rented out to many organisations and this is analogous to an inter-company or *competitive outcome*. Here a computational task going onto the UDC or a server currently owning a task that is purchasing resources will look to seek the cheapest price for resources from all of the willing servers in the auction. Each situation may well have a significant effect on the dynamics of the resource allocation.

4.3 Simulation Experiments

4.3.1 Illustrative Tests and Proof Of Concept

To demonstrate the viability of the distributed MBC system, initially a very simple setup is used (similar to that described with the example in Section 4.2.3)

where the UDC network constitutes 100 servers spread in a line with connectivity 5 to each side. As before, each server has a total of 10 neighbouring servers with circular boundary conditions.

Only a single server, in this instance server 50 of 100, is fed computational tasks at a rate of 1 task per timestep. Further constraints dictate that all tasks have a set load requirement of 1 resource unit, are of infinite duration, and that all servers have a total capacity of 1 resource unit. This means that a given server can only be running at maximum one task at a time. The limit prices of all agents were set within the bounds of \$0.75 and \$3.25, and non-competitive auction outcomes are used.

Smith's α parameter is recorded at each timestep of the simulation where trade(s) have occurred, as is the transaction price for every trade. To reiterate, the α parameter measures the root mean square deviation of the transaction price from the equilibrium price, and thus can provide, over several auctions, a measure of whether and to what degree the market is equilibrating and thus whether the relative worth of computational resources is being sought.

It is expected that as tasks 'drip-feed' into the local market of server 50, they will, over time, propagate out to neighbouring servers and eventually across the whole UDC network. However, as explained the speed and means by which the tasks are 'traded' across the network is expected to be determined largely by the policy used for task shifting.

4.3.2 Task-Shifting Policies

The first set of tests experiment with different task shifting policies in order to observe how this affects the dynamics of the task distribution across the UDC.

No Inter-Server Task Shifting

To begin with, the servers are not permitted to shift currently running tasks in order to partake in a new task auction under any circumstances. Intuitively this limits the coverage of a single task to the local neighbourhood of servers to which it enters the UDC. The simulation was run for 100 timesteps with the output visualisations and statistic plots shown in Figures 4.4 through 4.9.

Figure 4.4 shows that, as expected, the tasks entering server 50 spread to run both on this server and all neighbouring servers but not any further (by

the 100th timestep, 11 tasks are running). The server seller agent quote price plot of Figure 4.6 shows that after a very small number of timesteps, the quote price of the server seller agents rise rapidly (to a near maximum) as new tasks enter the local market of Server 50 and the demand for server resources steadily increases. As the neighbouring servers are not able to shift tasks, the server buyer agents are in effect never used to buy resources on behalf of their servers, and so the adjustment of the quote prices of these agents is minimal (Fig. 4.7).

To recap, in this test, all servers have a total resource unit capacity of 1. As the server seller agents' limit prices are set relative to this resource unit capacity, this creates a flat supply curve giving an equilibrium price of \$0.5. This may go some way in explaining the volatility of the transaction prices (Fig. 4.8) and α parameter values (Fig. 4.9). A much more likely reason is that, as computational tasks are only entering the local market of server 50 and as a result a limited number of auctions are going ahead (further exemplified by the no task-shift policy), the opportunities for the ZIP agents to adapt to other agents quotes is severely limited. With such a small number of auctions going ahead any convergence to the equilibrium price is not possible. This point is touched upon again with the experiments carried out in Section 4.3.3.

Over 50 trials within the 100 timesteps, the average percentage of tasks *allocated* onto the UDC network over all tasks that have *entered* the UDC (markets) was 11.0%.

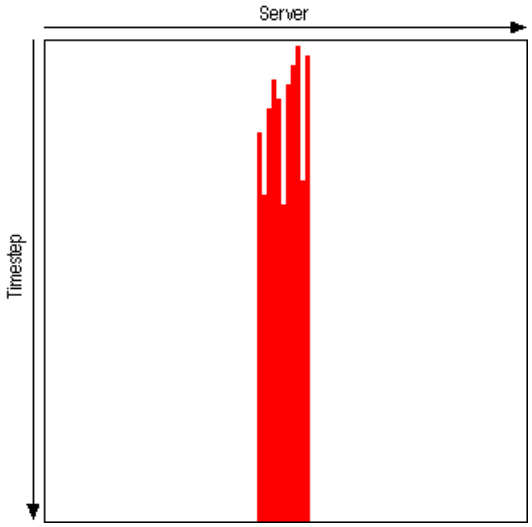

Figure 4.4: No task shifting: MBC simulation output visualisation of server load, with axes labels indicating meaning. A filled cell indicates that the Server is running a computational task.

Figure 4.5: Tasks in local market (same output for all tests in Section 4.3.2).

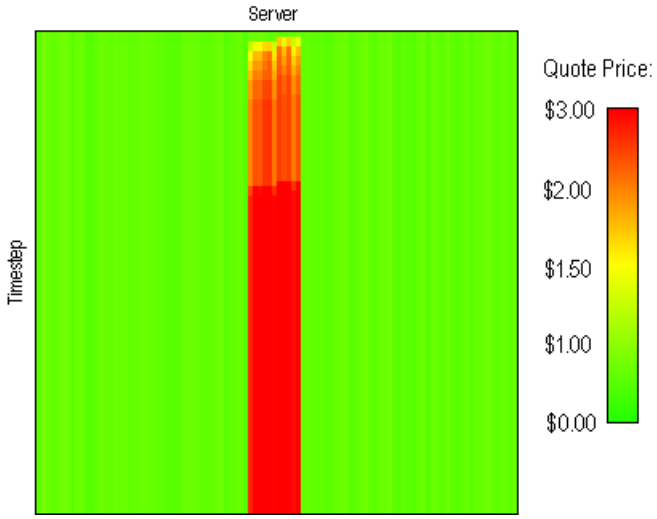

Figure 4.6: No task shifting: server seller ZIP agent quote price (with legend).

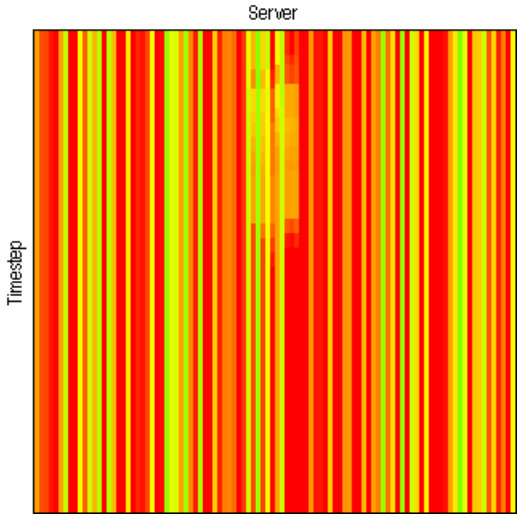

Figure 4.7: No task shifting: server buyer ZIP agent quote price.

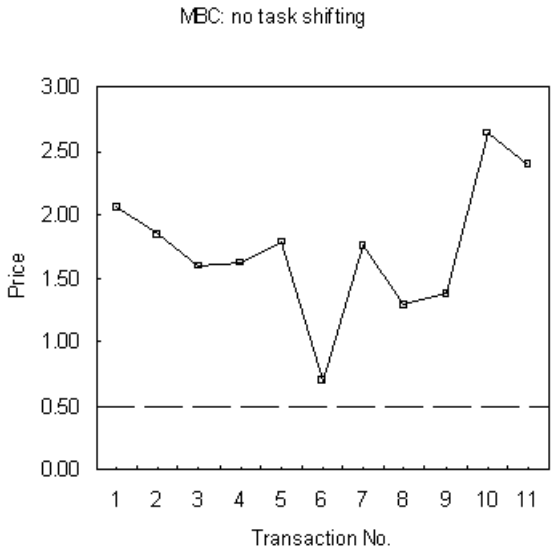

Figure 4.8: No task shifting: transaction prices.

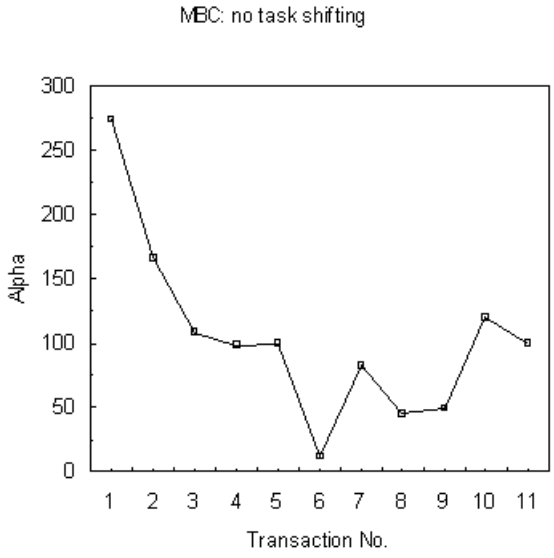

Figure 4.9: No task shifting: Smith's alpha.

Always Task Shift

For this test, the task-shifting policy was changed to dictate that, when an auction is initiated by a given server, all the neighbouring servers that have the *potential* capacity to participate in an auction (i.e. have enough total capacity but not enough available capacity) will endeavour to shift existing running task(s) onto one of *their* neighbouring servers, so in effect a 'sub' auction is initiated. As long as a neighbouring server is currently running an existing task, it will attempt to initiate an auction to offload it. This means that the original auction for the market task will not commence until all sub-auctions have been resolved (should there be any). In this test the simulation was run for 150 timesteps, with the output visualisations and statistic plots shown in Figures 4.10 through 4.14.

In analysis, the Server load plot (Fig. 4.10) shows the computational tasks entering the UDC quickly spreading out to neighbouring servers in order to make space for servers (local to Server 50) wishing to participate in auctions for new tasks. The neighbouring servers to the left of Server 50 especially indicate repetitive 'outward' shifting of groups of tasks. While the MBC simulation has been successful in initiating this task shifting behaviour, the apparent disadvantage is that there are large gaps where servers are not running tasks, or at least not running them for very long. This is due to the continual task shifting by servers in order to participate in new auctions. The server seller ZIP agent quote prices (Fig. 4.11) are seen to rise considerably in the area around Server 50 where the demand is clearly highest, although the quote prices tail off towards the end of the run. The plot also shows that the quote prices fall away rapidly from the servers not in the local neighbourhood of Server 50, and this is reflected in the server buyer ZIP agent quote price plot (Fig. 4.12) whose quote prices similarly fall, but again less so around Server 50.

As this policy of always shifting tasks has intuitively led to more auctions occurring across the UDC (some to first allocate tasks computational resources and others between servers shifting tasks), this has led to more opportunity for 'local' agents (in local neighbourhoods where successful / unsuccessful auctions occur) to adapt their quote prices accordingly. This is reflected in the transaction price (Fig. 4.13) and α parameter (Fig. 4.14) plots, where a (although still highly volatile) trend of equilibrium price ($0.50) convergence is shown, indicating a clear improvement over the 'no task shifting' policy of the previous

test.

Over 50 trials within the 150 timesteps, the average percentage of tasks allocated onto the UDC network over all tasks that have entered the UDC was 26.6%.

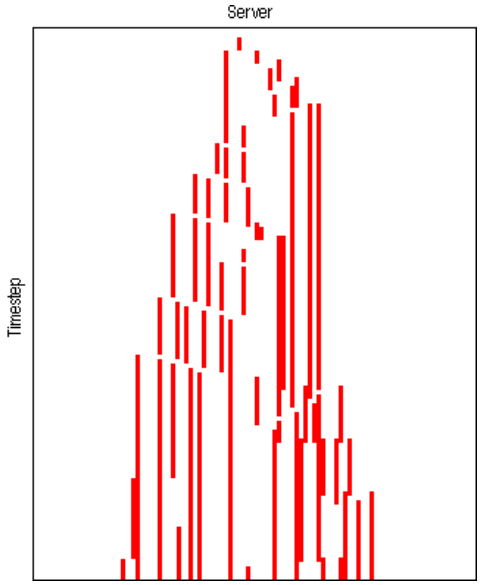

Figure 4.10: Always task shift: server load.

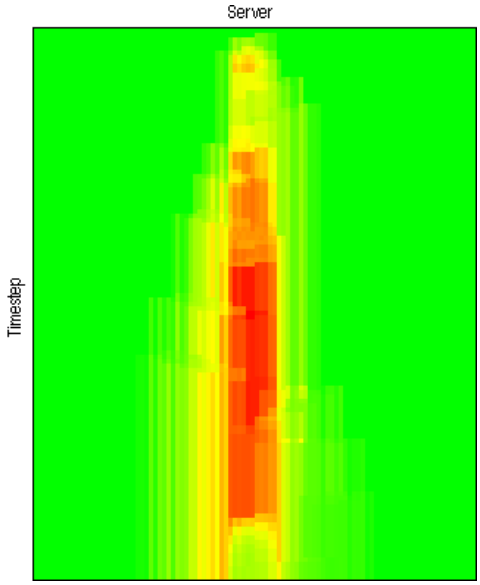

Figure 4.11: Always task shift: server seller ZIP agent quote price.

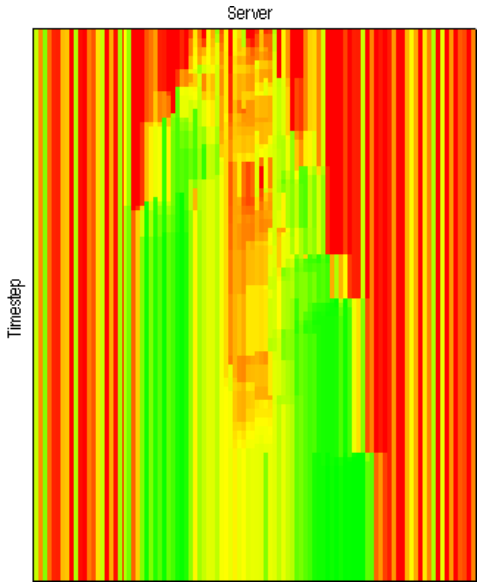

Figure 4.12: Always task shift: server buyer ZIP agent quote price.

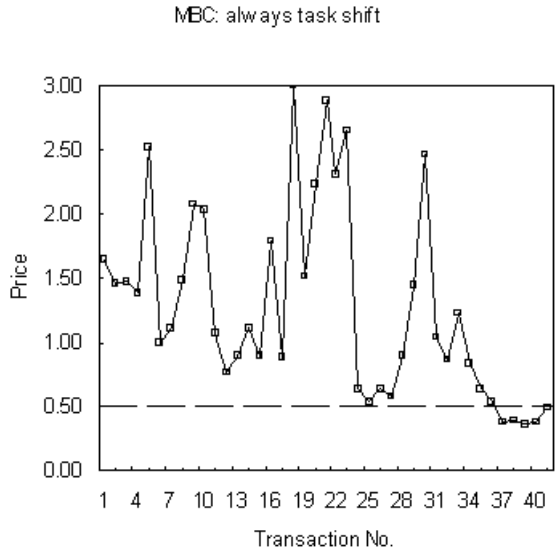

Figure 4.13: Always task shift: transaction prices.

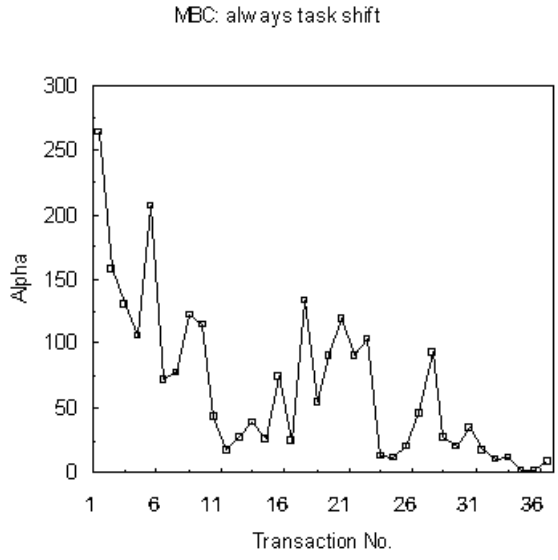

Figure 4.14: Always task shift: Smith's alpha.

Randomly Shift One Task

For this test, the task shifting policy proceeded as follows: given an auction request by a server, all neighbouring servers would initially indicate their willingness to participate *only* if they had enough available capacity. If the case occurs that none of them can participate due to them all currently running tasks (which is very likely to occur given anything more than moderate task traffic), a randomly selected neighbouring server is then selected to attempt to offload a task. As with the other task-shifting policies, this is continued over as many server 'hops' as is necessary. This gives a reasonable chance that a new auction can go ahead (regardless of whether or not it is successful), but does not 'upset' large numbers of neighbouring servers by spending valuable timesteps offloading their currently running tasks.

Thus the idea behind this policy is to 'squeeze' new tasks onto the UDC network with the minimum of disruption. If traffic on the network is very high, however, and a free server cannot be found after a pre-set number of timesteps, the auction for the new task is terminated. In this test the simulation was run for 150 timesteps, with the output visualisations and statistic plots shown in Figures 4.15 through 4.19.

In comparison to the 'always task shift' policy, the server load plot (Fig. 4.15) indicates a more progressive spread of tasks to neighbouring servers, where it can be seen that servers running tasks in the local neighbourhood of Server 50 have been selected to offload tasks in order to participate in auctions for new computational tasks. This behaviour has led to several rises and falls in the quote prices of server seller (Fig. 4.16) and server buyer ZIP agents (Fig. 4.17) over the 150 timesteps. This could be the result of equal numbers of successful and unsuccessful auctions as it certainly does not indicate an increase in demand for the server resources (as indicated in the earlier quote price plots of Figures 4.6 and 4.11). While the transaction prices of this particular simulation run (Fig. 4.18) are, as with the other simulation tests thus far, relatively volatile (due to the agents adapting to quite a small number of auctions), an overall trend towards the equilibrium price of $0.50 has still been made (Fig. 4.19).

Over 50 trials within the 150 timesteps, the average percentage of tasks allocated onto the UDC network over all tasks that have entered the UDC was 21.4%.

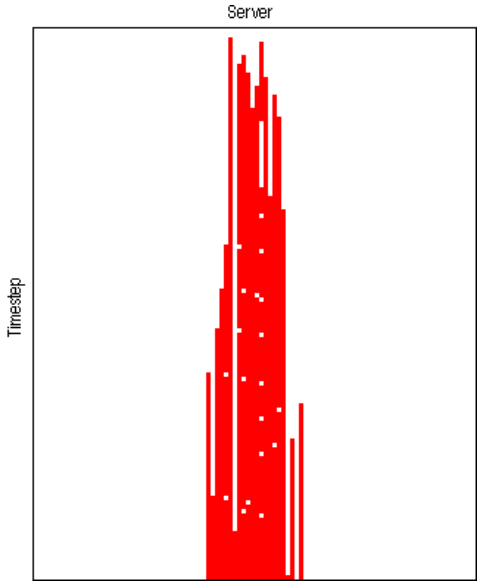

Figure 4.15: Randomly shift one task: server load.

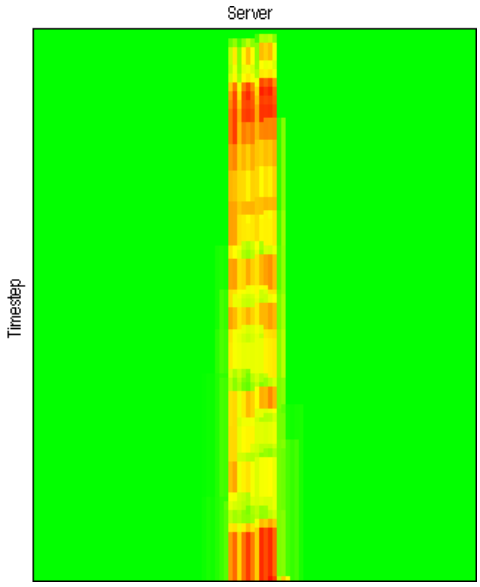

Figure 4.16: Randomly shift one task: server seller ZIP agent quote price.

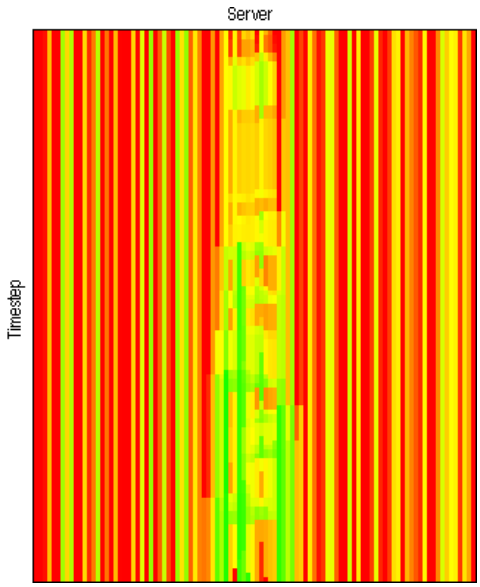

Figure 4.17: Randomly shift one task: server buyer ZIP agent quote price.

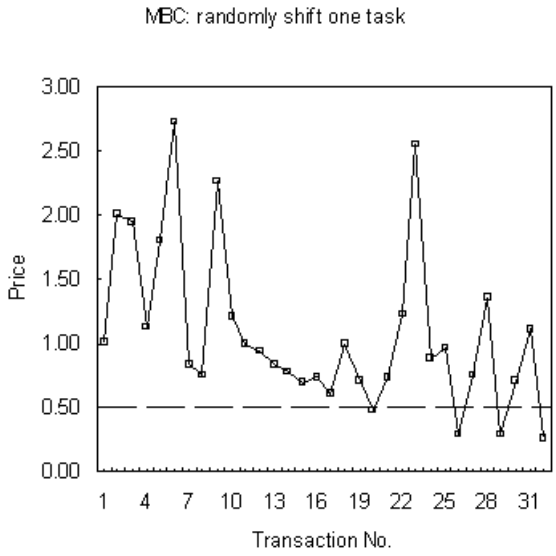

Figure 4.18: Randomly shift one task: transaction prices.

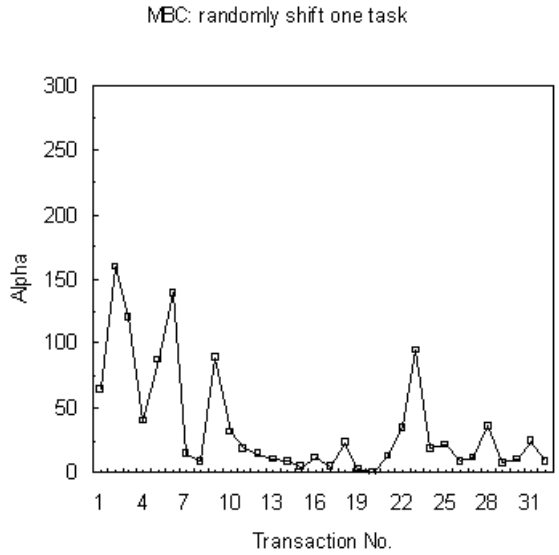

Figure 4.19: Randomly shift one task: Smith's alpha.

4.3.3 Varying Task Load Requirements and Server Capacities

In all tests carried out so far, a number of simplifying constraints have been applied – for the computational tasks entering the UDC network for allocation, all task resource unit requirements have been set to one, and the duration of the tasks set as infinite (i.e. to beyond the length of the simulation running time). For the servers on the UDC network, their maximum resource unit capacity has been restricted to one. These constraints have meant that a given server could only run a single computational task at a time.

For this set of simulation runs, these constraints are removed to allow for tasks that have varying resource requirements and durations, and servers that have varying levels of total resource unit capacity, thus making the resource allocation process more challenging. As the limit prices for the ZIP agents are dependent upon such factors (Section 4.2.4), the network wide equilibrium price is now set at a level around $2.00. A comparison of the resource allocation performance over 50 trials is now made between the 'always task shift' and

'randomly shift one task' task shifting policies.

Always Task Shift

In this experiment, computational tasks entering the UDC network for allocation can randomly enter the local market of *any* server. Now the MBC system must allocate resources for tasks across the entire network. In particular, for every server on the network, a new task has a 10% chance of entering its local market at each timestep, which gives enough task traffic for swift network saturation.

Figure 4.20 shows a sample server load plot over 150 timesteps using the 'always task shift' task shifting policy – the rather scant allocation of tasks onto the network is somewhat confirmed by a task allocation percentage of 17.1% (averaged over 50 simulation trials). This can be explained by the continual task shifting of tasks already running on the network effectively preventing the initiation of auctions for new tasks waiting in the local markets of servers.

One of the points made in the analysis of Section 4.3.2 tests concerned the market equilibration of the ZIP agent quote prices, and how the limited number of auctions brought about by 'drip feeding' tasks into the local market of Server 50 caused a very volatile and weak convergence trend towards the equilibrium price of $0.50. In this test, as computational tasks are entering local markets and being allocated across the entire network, the ZIP agents across the network are given many opportunities to locally adapt their quote prices to successful and unsuccessful quote prices. This hypothesis is reflected in the α statistic plot (Fig. 4.21) for a sample trial – a much less volatile and closer convergence towards the equilibrium price is made, even though (as expected) it changes slightly over the course of the simulation run (Fig. 4.22).

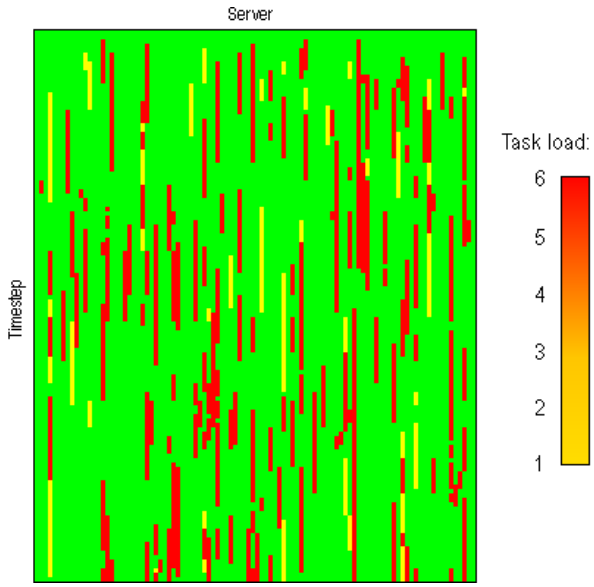

Figure 4.20: Always task shift, with variation of task load requirements, duration and server load capacity: server load.

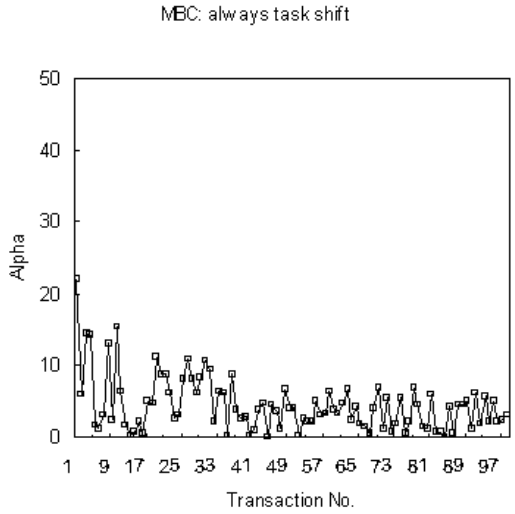

Figure 4.21: Always task shift, with variation of task load requirements, duration and server load capacity: Smith's alpha.

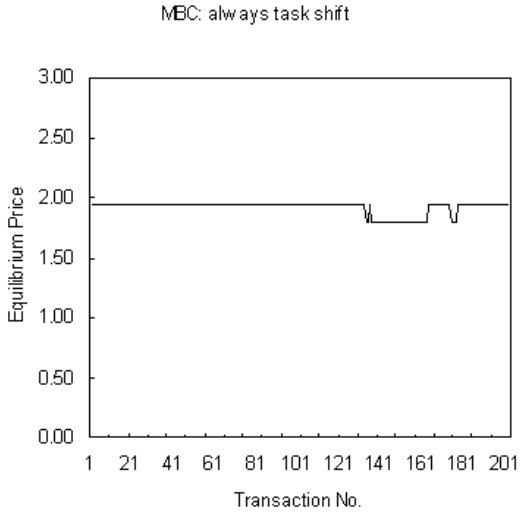

Figure 4.22: Always task shift, with variation of task load requirements, duration and server load capacity: equilibrium price.

Randomly Shift One Task

An improved resource allocation performance is shown by the 'randomly shift one task only' task shifting policy (Fig. 4.23). Using this policy the resource allocation performance over 50 trials was 24.1%. The minimal 'disruption' caused by this task shifting policy obviously pays off when used across the entire network, which is contrary to the situation of tasks singularly entering Server 50, where it was seen that the 'always task shift' gave better resource allocation performance (Section 4.3.2). However, the α plot for a sample trial (Fig. 4.24) gives a very similar convergence trend to that in Fig. 4.21.

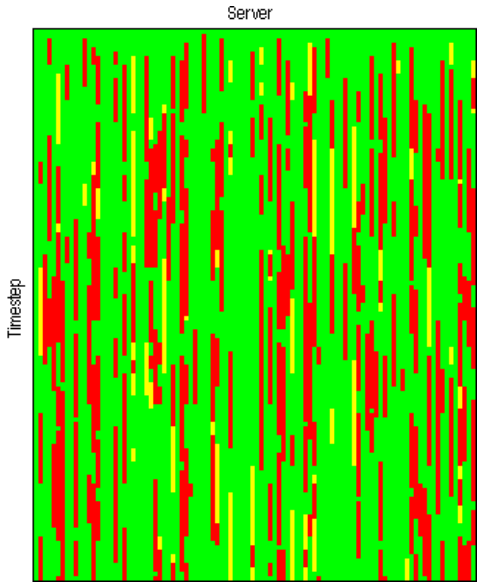

Figure 4.23: Randomly shift one task, with variation of task load requirements, duration and server load capacity: server load.

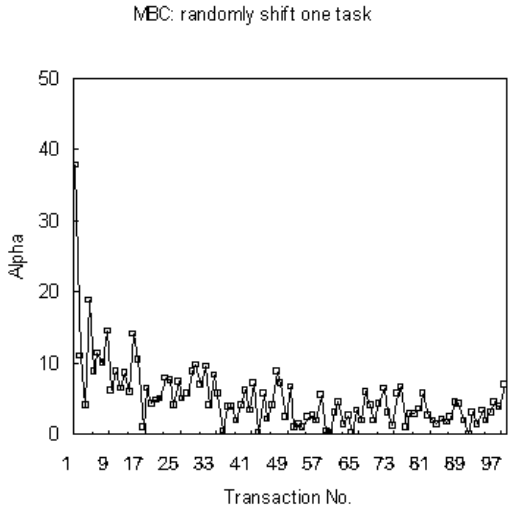

Figure 4.24: Randomly shift one task, with variation of task load requirements, duration and server load capacity: Smith's alpha.

4.3.4 Competitive and Non-Competitive Outcomes

The 'Closing Auction Deals' part of Section 4.2.4 explained the two means by which a given ZIP agent involved in a successful trade can close an auction deal – by either randomly selecting a willing server (termed a *non-competitive outcome*) or by specifically selecting the server giving the best price (termed a *competitive outcome*). The purpose of this final set of tests is to see whether either outcome will effect the dynamics of the MBC resource allocation, and the convergence of the ZIP agents towards the equilibrium price. Two sets of 15 MBC simulation trials were run (each of 150 timesteps), one set using the non-competitive outcome and the other the competitive outcome. As for the tests in the previous section (4.3.3), no constraints were applied to the server capacities nor task resource requirements, and computational tasks were allowed to enter the local market of all UDC network servers.

The average transaction price and α plots are shown in Figures 4.25 through 4.28. It is clear from these plots that the market equilibration behaviour is near to identical when using either non-competitive or competitive outcomes. Add to this the similarity in transaction prices (Figures 4.25 and 4.26), and it is concluded from this experiment that there is no difference in the dynamics of the resource allocation when using either outcome.

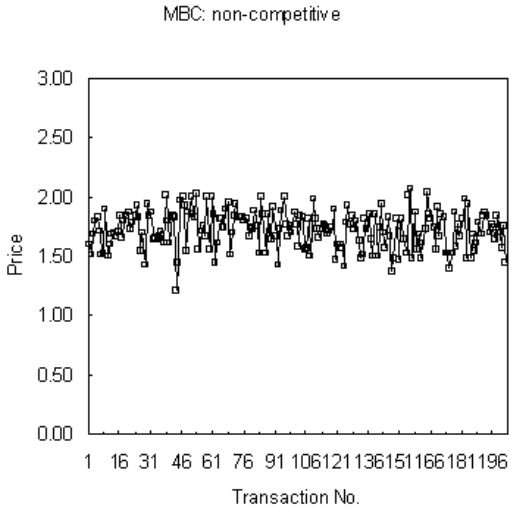

Figure 4.25: Non-competitive outcome: transaction prices averaged over 15 MBC trials.

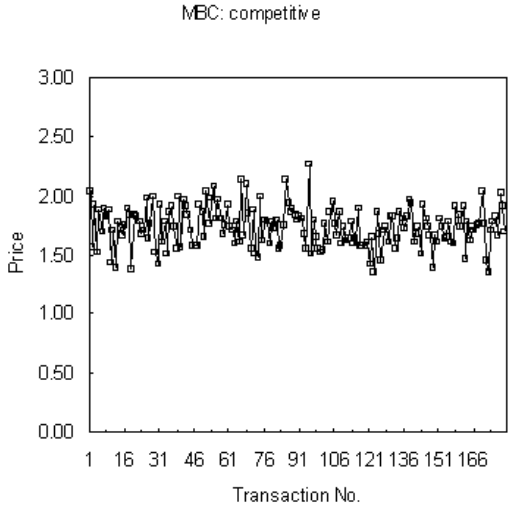

Figure 4.26: Competitive outcome: transaction prices averaged over 15 MBC trials.

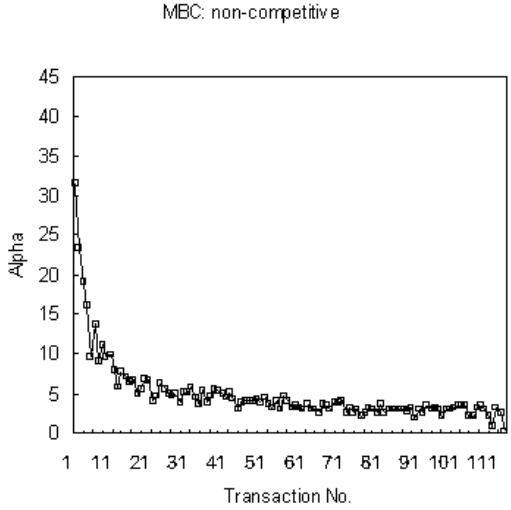

Figure 4.27: Non-competitive outcome: Smith's alpha averaged over 15 MBC trials.

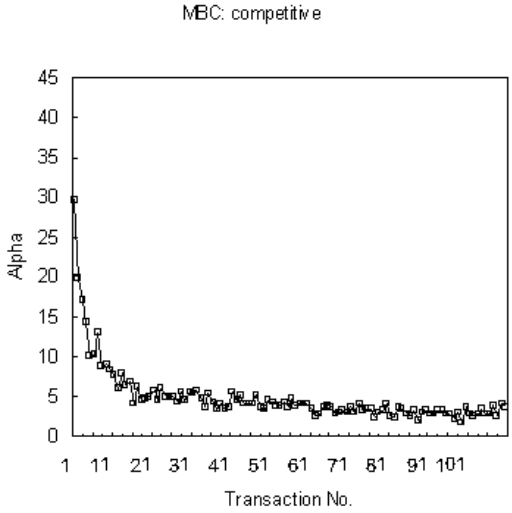

Figure 4.28: Competitive outcome: Smith's alpha averaged over 15 MBC trials.

4.4 Summary

The presented results have shown that the MBC simulation has been successful in efficiently allocating compute resources to the computational tasks that come onto the market, and in trading them server-to-server across the UDC. However, whether the design of this particular MBC simulation is comparable to that of other such systems is definitely something that would be of use in judging its overall 'performance' even though to date (as discussed in Chapter 2), there are still very few distributed and autonomous systems such as the one developed here.

Additionally, the tests have highlighted that the means by which the tasks are allocated (and the number that are allocated) is greatly affected by a multitude of parameters, some less so than others. The task-shifting policies introduced in this chapter no doubt have a very large effect – even the optimality of each policy can vary depending on the amount of computational task traffic coming onto the network. For example, the 'always task shift' policy proved to allocate tasks onto the network most efficiently when tasks were only entering the local market of Server 50, but when they were coming from all over the UDC network, the 'randomly shift one task' policy proved more effective.

Thus it is concluded that, just as for the ZIP trading-agent simulation before it, the MBC simulation has many parameters that could, using an evolutionary algorithm, be fine-tuned in order to hone the resource allocation efficiency and market equilibration performance of the system. This is now explored in the next chapter.

Chapter 5

Evolutionary Optimisation of the MBC System

5.1 Introduction

As was explained in the summary of Chapter 4, the resource allocation and market equilibration performance of the MBC simulation is dependent on the settings of multiple parameters such as, for example, those dictating the operation of the task-shifting policy used, and how many timesteps an auction-initiating server should wait for every neighbouring servers ability to partake in a proposed auction. It is therefore postulated that the particular values of these parameters can have a significantly fundamental effect on both the performance and overall (macro-scale) behaviour of the system. This is an important issue given that, without complete and exhaustive testing, the settings of these parameters cannot be optimally determined *a priori*. For this reason, it is apparent that use of a GA to evolve particular (sets of) parameters within the MBC simulation would help in fine-tuning system performance.

Thus, in a similar experimental setup to that described in Chapter 3, this chapter describes the use of a GA to evolve the ZIP trading-agent parameter sets acting within the MBC simulation, but with the added addition of a parameter dictating the particular market mechanism that the agents operate within.

5.2 The Evolution of Market Mechanisms

In all of the experiments carried out so far with both the ZIP simulation and the MBC simulation, the type of auction that the ZIP agents have been interacting within has been that of a continuous double auction (CDA). In the CDA, sellers quote decreasing offer prices while at the same time the buyers are making quotes of increasingly higher bid prices, and where at any time the sellers are free to accept any buyer's bid and the buyers being free to accept any seller's offer. However, it cannot be proven that use of the CDA market would give the most desirable dynamics, or best allocative efficiency. As reviewed in Section 2.4.2, this was a point was first brought to the fore by Cliff [18, 19], when it was suggested that the market mechanism be put under evolutionary control in order to (perhaps) seek markets that give better dynamics than that of the CDA.

These 'other' markets exist on a continuum of possible auction types dictated by a parameter Q_s [18], which determines the *probability that a seller*

quotes within a given timeslice of the market (noting that, if Q_b represents the probability that a buyer quotes within a timeslice, as Q_b = 1.0 - Q_s, it is only necessary to determine Q_s [18]). So for a CDA market, Q_s would equal 0.5 (i.e. a quote is equally likely from each side), for an English Auction (where only buyers quote prices), $Q_s = 0.0$ and for a Dutch Auction (where only sellers quote), $Q_s = 1.0$. However, as explained in [18], in between these human-designed marketplaces is a space of market types where it is conceivable that Q_s could take on *any possible value*. The details of using a GA to evolve the Q_s parameter along with the ZIP-trading agent parameter sets is now described.

5.3 Experimental Details

5.3.1 Evolutionary Algorithm and Encoding Scheme

The evolutionary algorithm employed for all of the experiments carried out in this section is exactly the same as that presented in Section 3.4.1, and so will not be described further here. The encoding scheme used for the genome, however, differs in two respects from that used in Chapter 3 – firstly, two sets of ZIP agent adaptation parameter sets are specified, one set for the buyer ZIP agents in the MBC simulation, and another for the seller ZIP agents in the MBC simulation. Secondly, the addition of the Q_s parameter on the genome.

Thus, the genome now consists of the following 17-real valued parameters by which the adaptation of the ZIP trader market is determined: the 3 pairs of bounds on the distribution of the (learning rate β_i, momentum γ_i and initial profit coefficient $\mu_i(0)$) parameters for the individual agents ($\beta_b, \beta_\Delta, \gamma_b, \gamma_\Delta, \mu_b$, and μ_Δ), the two parameters c_r and c_a (see Section 3.2.2) that define the distributions of the stochastic perturbations used in calculating each agent's target price, and of course the Q_s parameter. As a vector V, this is defined as follows:

$$V = [\beta_{bb}, \beta_{\Delta b}, \gamma_{bb}, \gamma_{\Delta b}, \mu_{bb}, \mu_{\Delta b}, c_{rb}, c_{ab}, \beta_{bs}, \\ \beta_{\Delta s}, \gamma_{bs}, \gamma_{\Delta s}, \mu_{bs}, \mu_{\Delta s}, c_{rs}, c_{as}, Q_s] \in \Re^{17}$$

where the second appended subscript 'b' denotes a buyer parameter and 's' denotes a seller parameter. For all experiments, the ZIP agent adaptation parameters are initially set (as in Chapter 3) to the V_{cb} values, which were originally hand-picked to give reasonable market performance. Add in the Q_s

parameter, which is set to 0.5 in order to resemble a CDA market, and the following parameter vector, denoted V_{mbc}, results:

V_{mbc} = [0.1,0.4,0.00,0.1,0.05,0.3,0.05,0.05,0.1,0.4,0.00,0.1,0.05,0.3,0.05,0.05,0.5]

5.3.2 Fitness Evaluation Scheme

The fitness of each genome V_i was calculated as the mean average of the Smith's alpha statistic for every transaction that occurred across the UDC within the specified number of timesteps, over the number of independent genome MBC trials (see next section). Formally:

$$F(V_i) = \frac{1}{n} \sum_{n=1}^{n} \frac{1}{t} \sum_{t=1}^{t} \alpha_i(t)$$

where n is equal to the number of MBC trials and t the number of transactions within a single trial. Finally, as before, the GA is attempting to *minimise* the fitness score, with the optimum being $F(V_i) = 0.0$.

5.3.3 The Experimental UDC Network

The UDC network was setup to contain 100 servers with (as in the experiments carried out in Chapter 4) each server having a connectivity 5 to each side. The 'always task shift' policy was used, with computational tasks able to enter the local markets of all servers on the network, and using variable task load requirements and server capacities. The limit prices of all agents on the network were set in accordance of the description given in Section 4.2.4, within the ranges of \$0.75 to \$3.25, to give an equilibrium price on or around \$2.00.

Each experiment consisted of 8 independent evolutionary runs of 100 generations each, within which each genome, in a population of 30 genomes, was evaluated on the average of 20 MBC trials. Each trial consisted of a 100 timestep run of the MBC simulation.

5.4 Results

5.4.1 Evolving the ZIP Agent Parameter Sets along with the Q_s Parameter

The first plot (Fig. 5.1) shows the fitness of the elite genome at every generation for each of the 8 evolutionary runs. It indicates that the GA was able to steadily improve the market equilibration performance of the ZIP trading agents, but after 30 generations the mean alpha fitness score levelled out. However, the final fitness scores (in the range of around 2.8 to 4.0) show a performance easily comparable to that found by the GA evolving the ZIP agents operating in basic markets (Fig. 3.16).

The plot of Figure 5.2 shows the evolved value of the Q_s parameter at each of the 100 generations, again for all 8 runs. Interestingly, the sharp evolutionary trajectory indicates that the MBC simulation has given favourable market equilibration performance to markets where the Q_s value is set at 0.0. This corresponds to the (human designed) English Auction, where only the buyers quote prices. Fig. 5.3 shows a sample MBC simulation trial transaction price time-series at the final generation of the 'best' evolutionary run (where $Q_s = 0.0$), together with the underlying equilibrium price, and Fig. 5.4 the accompanying α plot. In comparison with a sample α plot taken after 0 generations (see Fig. 4.20), it can be seen that there is much less volatility in the transaction prices, confirming that the market equilibration has improved considerably.

In light of these results, a further two sets of experiments were carried out: one where the Q_s was 'clamped' to 0.0, and another where Q_s is set to 0.5. The purpose of setting Q_s to 0.0 is to see whether the GA can find a fitness improvement from the value converged upon in this first set of experiments, and when Q_s is equal to 0.5 (the CDA value), to indicate if a value of 0.0 is indeed optimal. The experiments where Q_s is set to 0.5 are reported on first.

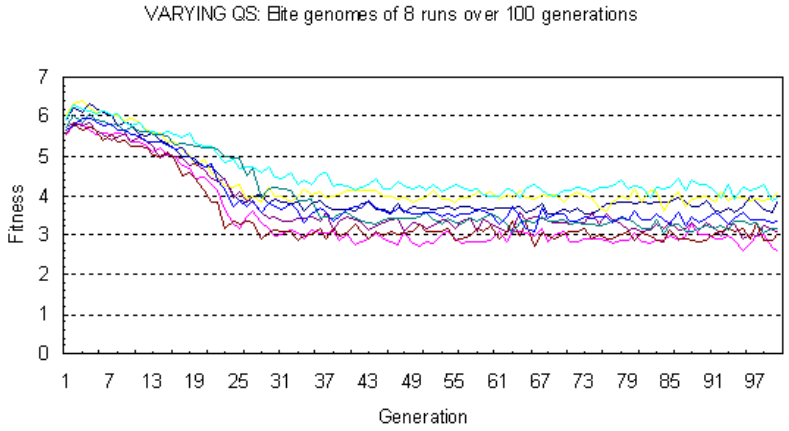

Figure 5.1: Elite genomes of 8 runs over 100 generations with evolution of the Q_s parameter.

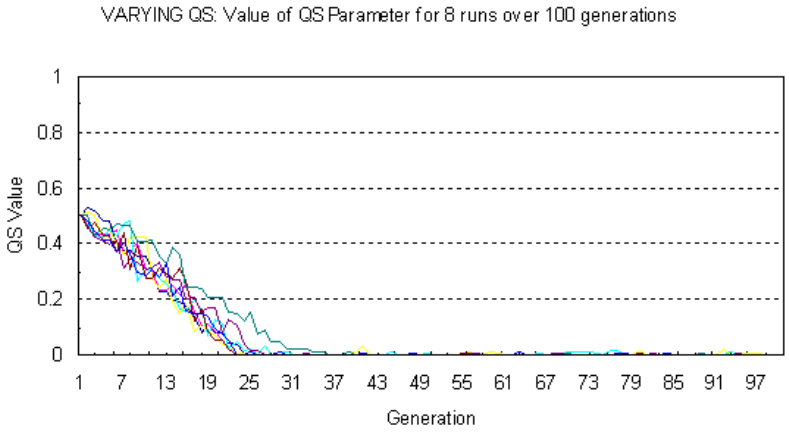

Figure 5.2: Value of the Q_s parameter for each of the 8 runs over 100 generations.

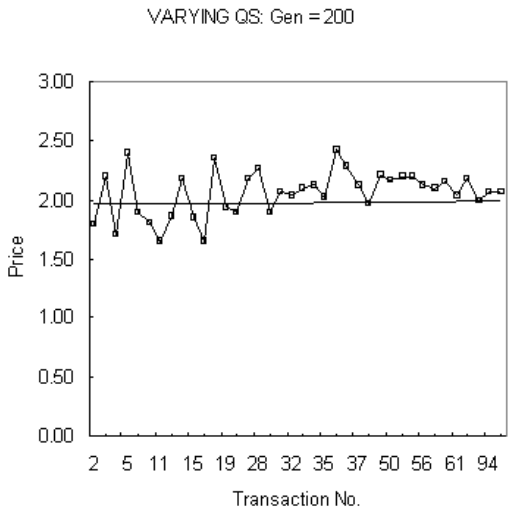

Figure 5.3: A sample transaction price time-series (with underlying equilibrium price) from one trial of the final generation of the 'best' evolutionary run, when evolving the Q_s parameter.

5.4.2 Evolving ZIP Agent Parameter Sets with the Q_s Parameter set to 0.5

For this set of evolutionary runs, only the separate sets of buyer and seller ZIP agent parameter sets were evolved, as the value of Q_s was clamped to 0.5, i.e. that of a CDA market. Figure 5.5, a plot of the elite genomes at each of 100 generations for the 8 runs, shows that, when the MBC simulation uses CDA markets, the GA is unable to find market equilibration dynamics as desirable as that found when Q_s is allowed to take on any value (Fig. 5.1). Figure 5.6 shows a plot of a sample MBC simulation trial transaction price series at the final generation of the 'best' evolutionary run – in comparison with that of Figure 5.3, it is extremely volatile and not many transactions are closed around the equilibrium price. This volatility is also reflected in the α statistic plot of Fig. 5.7.

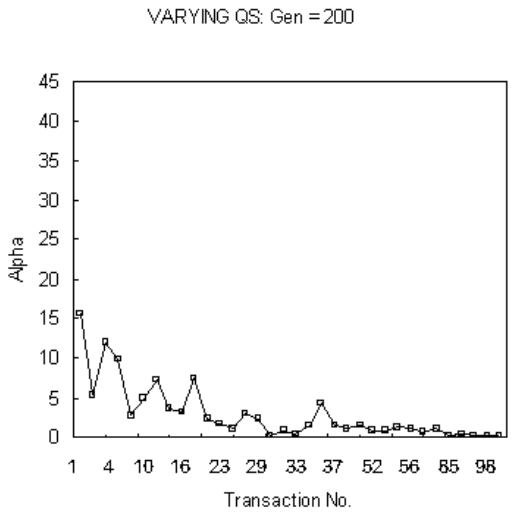

Figure 5.4: A sample alpha parameter statistic plot from one trial of the final generation of the 'best' evolutionary run, when evolving the Q_s parameter.

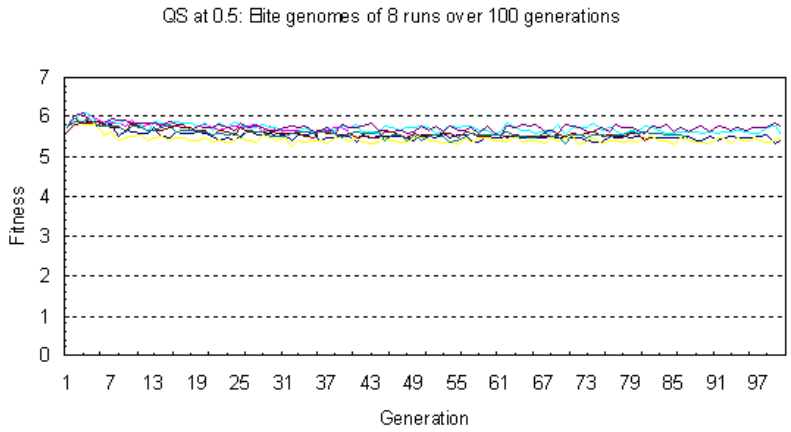

Figure 5.5: Elite genomes of 8 runs over 100 generations with Q_s parameter set at 0.5.

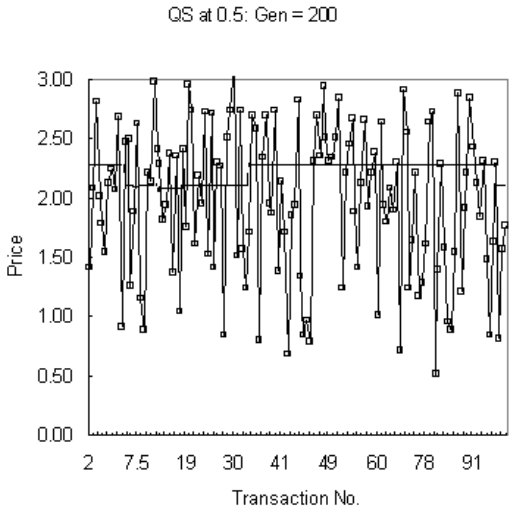

Figure 5.6: A sample transaction price time-series (with the underlying equilibrium price) from one trial of the final generation of the 'best' evolutionary run, when Q_s is set at 0.5.

5.4.3 Evolving ZIP Agent Parameter Sets with the Q_s Parameter set to 0.0

For the final set of evolutionary runs, the value of Q_s was clamped to 0.0, to resemble an English Auction. Figure 5.8, the plot of the elite genomes at each of 100 generations for the 8 runs, shows that the GA evolves fitness scores to the same level of that found when varying Q_s (Fig. 5.1), although converges to it faster by virture of the fact that the Q_s is already at 0.0 – the value at which we have already seen the GA evolve to in Section 5.4.1. The plots of both the sample MBC simulation trial transaction price series (Fig. 5.9)and the α statistic (Fig. 5.10) at the final generation of the 'best' evolutionary run confirms that a Q_s of 0.0 must be optimal by showing the exhibiting the best market equilibration behaviour of all the experiments. However, although not considered here (see next section), the ZIP trading agent parameters were also evolved and thus it is concluded that a combination of the final generation parameter set together with Q_s equalling 0.0 gives the most desirable market dynamics.

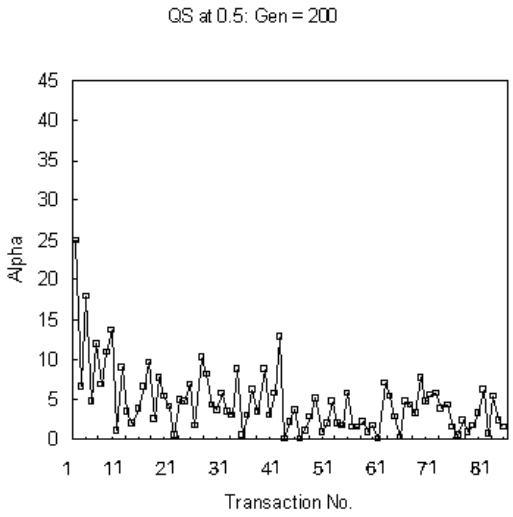

Figure 5.7: A sample alpha parameter statistic plot from one trial of the final generation of the 'best' evolutionary run, when Q_s is set at 0.5.

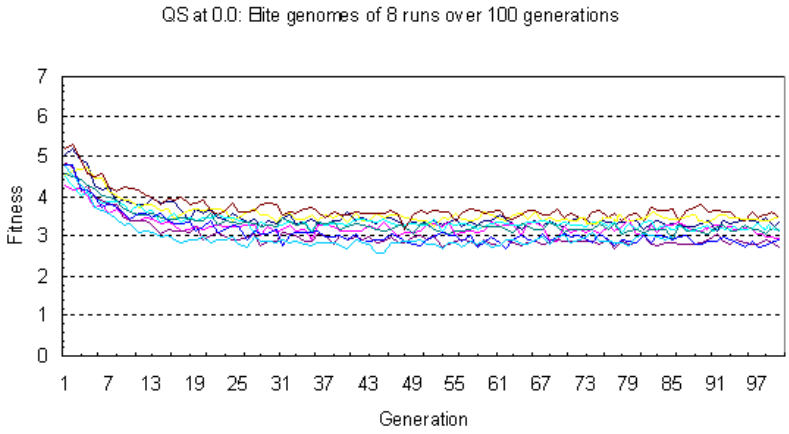

Figure 5.8: Elite genomes of 8 runs over 100 generations with Q_s parameter set at 0.0.

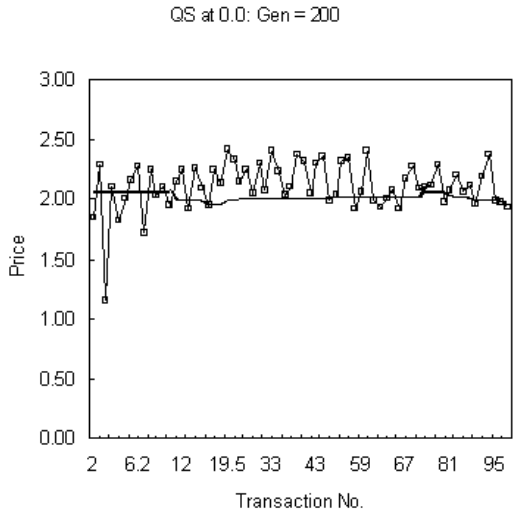

Figure 5.9: A sample transaction price time-series (with underlying equilibrium price) from one trial of the final generation of the 'best' evolutionary run, when Q_s is set at 0.0.

5.4.4 An Analysis of the Evolved ZIP-Agent Parameter Sets

Finally, a brief analysis is made of the ZIP agent parameter sets evolved by the evolutionary run (over all three experiments) giving the best fitness score at the final generation. This was identified to be Run 4 in the experiment where the Q_s value was set at 0.0. To aid a comparison of the separate buyer and seller ZIP agent parameter vectors evolved in this run, they are split into two vectors, denoted V_{buyer} and V_{seller}, as follows:

$$V_{buyer} = [0.341,0.076,0.000,0.023,0.255,0.013,0.192,0.111]$$

$$V_{seller} = [0.247,0.315,0.072,0.427,0.227,0.523,0.081,0.366]$$

As can be seen, there is a marked difference in most of the parameter settings for the buyer and seller agents – in general, the ZIP seller agents parameters are generally higher than that of the ZIP buyer agents. This indicates that the seller agents have been evolved to adapt more rapidly to the buyer agent quote prices

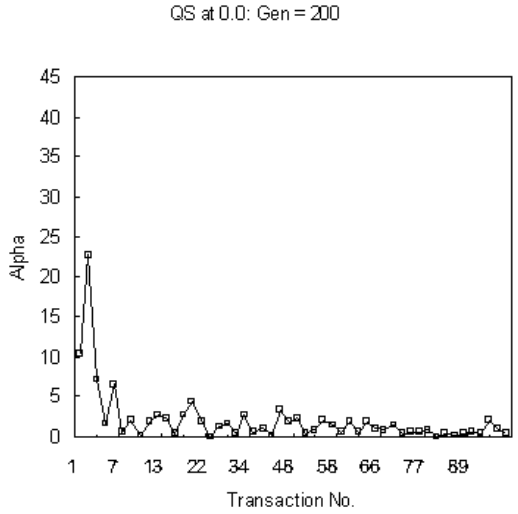

Figure 5.10: A sample alpha parameter statistic plot from one trial of the final generation of the 'best' evolutionary run, when Q_s is set at 0.0.

in the market, which makes sense given that the 0.0 Q_s value (i.e. that earlier evolved to by the GA) is a market where only the buyers quote prices. Thus, the GA has taken advantage of the ability to evolve separate buyer and seller parameter sets by 'moulding' them to work well within a $Q_s = 0.0$ market.

5.5 Summary

The results presented in this chapter have demonstrated that the market equilibration behaviour of the MBC simulation can be improved by placing both the ZIP trading agent parameter sets and a parameter dictating the market mechanism, under evolutionary control. In summary, the convergence towards the transaction price was most marked when only the buyer ZIP agents in the market were able to quote prices.

Chapter 6

Future Work and Conclusions

6.1 Future Work

Even though the project has been successful in attaining its original aims, there are many areas of development and further paths of investigation that, given more time, could be explored. A selection of the more pertinent and promising avenues for future work are summarised as follows:

- **Further experiments and exploration with the MBC simulation**: the tests reported on in Chapter 4 have only touched the surface concerning (for example) the experimentation with different parameter values and the means by which computational tasks are traded across the UDC network. Other areas of potential investigation include how the system would react to server node and / or connectivity failures, and the use of different server connectivities and neighbourhood functions.

- **Modelling effects of noise or information delays within the system**: looking at how the resource allocation and market equilibration behaviour of the ZIP agents could be affected by locally imperfect or delayed information (in similar lines to the original work by Hogg and Huberman [20] on the dynamics of large distributed computational systems).

- **Use of co-evolutionary techniques to co-evolve the ZIP trading agent parameters and computational task schedules**: in a similar vein to Hillis' work on host-parasite co-evolution of sorting networks [21], a possible area for further investigation could be to co-evolve the performance of the ZIP trading agents with specific computational task schedules (i.e. a real time stream of computational tasks). A more rapidly fine-tuned MBC could result, as the parasites that show up weaknesses in the MBC simulation would be widely represented in their population.

- **Increasing the realism and complexity of the simulated UDC**: the difficulty of the MBC resource allocation could be increased by introducing more specific task constraints – for example, by having the process modelled around the Resource Allocation Problem (RAP, [22]) which takes into consideration very specific computational task requirements and requires that bandwidth constraints on the network are satisfied while the communication delays between the assigned servers is minimised. The UDC in this case would consist of a hierarchical tiered tree of servers.

6.2 Conclusions

In summary this project has developed and reported on the use of a ZIP-based market-based control system for allocating computational resources on a UDC, and evolved ZIP agent parameter sets and marketplaces in order to fine-tune the performance of the system.

To firstly gain an understanding of the ZIP trading agents originally developed by Cliff [3], a re-implementation of the ZIP simulation was tested in a variety of basic market scenarios that demonstrated the agents' ability to adapt to market quote prices and converge to an equilibrium price. A GA was then employed to evolve the ZIP trading agent parameters and to improve on the initial ZIP agent market equilibration behaviour.

Once this ZIP trading agent grounding had been established, a MBC simulation was designed and implemented, the first of its kind to utilise ZIP trading agents within topologically distributed markets and for the purpose of performing computational load balancing and scheduling of compute tasks over a UDC. Control experiments indicated that the simulation was successful in being able to efficiently allocate and trade tasks among a network of UDC servers. These results have justified the ability for a MBC system such as this to provide a fully distributed, autonomous, and lightweight solution to load balance computational tasks in a UDC, and, indeed, open new avenues for MBC in a plethora of similarly complex control problems.

So, to summarise, this thesis presents the first results ever from the use of a GA to optimise the parameter values and market mechanism for ZIP traders in a UDC MBC context. Although the results are preliminary, the fact that (as shown in Section 5.4.4) nonstandard values of Q_s are found to be useful because the parameter vectors for buyers and sellers have evolved to different values is likely to be significant and deserves exploration in less minimal UDC simulations.

Appendix A

Background Economics

A.1 Introduction

This appendix gives a brief overview of the economic principles underlying market-based control. The first part deals with *microeconomics* which describes how supply and demand for a given commodity functions in a typical market. The second part briefly describes Vernon Smith's pioneering work in experimental economics.

A.2 Microeconomics

In succinct terms, microeconomics deals with the allocation of scarce resources, or, put another way, how individuals choose among particular commodities. The central concept within the standard economic framework is that of a *market* containing *buyers* (consumers) and of *sellers* (producers) of arbitrary commodities. The interaction of buyers and sellers within the market introduces the additional concepts of *supply*, *demand* and that of *equilibrium*.

A market is defined by Begg [7] as 'a set of arrangements by which buyers and sellers are in contact to exchange goods or services'. As reviewed in Section 2.3.2, there are a number of different market structures that the buyers and sellers trade within, although all perform essentially the same economic function.

A.3 Supply and Demand

The supply of a good refers to the quantity of a given commodity that the sellers in the market are willing to sell (i.e. the behaviour of the sellers) at each possible price. Conversely, the demand for a good refers to the quantity that the buyers are willing to purchase, or the behaviour of the buyers, at each possible price. In more specific terms, one can talk about a quantity supplied and a quantity demanded at each price of the commodity.

Intuitively it can be stated that as the price of a commodity rises, the supply rises but the demand falls, and if the price falls the demand rises but the supply falls. For example, sellers would wish to sell a good at a high price in order to get an increased profit on each unit but buyers would not wish to purchase the good at a high price because they only have limited funds in which to spend. However, as supply needs a demand and, equally, demand needs a supply, given

a high price (or a low price) there will be an *excess supply* (or *excess demand*). This is where the *equilibrium price* comes in as this is the intermediate price in which the supply equals the demand and the market *clears*. In other words, there is no excess supply or demand.

The mechanism by which markets achieve this equilibrium price point is a by-product of the participants desire to increase their own profit margin and to obtain the best possible deal. The incentive to change prices remains whenever there is an excess supply or an excess demand in the market and so over time it will approach the equilibrium price. In this sense the market is *self-correcting*. The equilibration that arises through the forces of supply and demand occurs naturally within a free-market, and so external price-fixing (for example by government regulation) would seriously disrupt the market because in reality the buyers do not have to buy and the sellers do not have to sell.

A.3.1 The Supply and Demand Curves

The relationship between the supply and demand of a good and its price can be illustrated by supply and demand curves (see Fig. A.1), where the intersection of the curves represents the equilibrium price. The vertical axis on the graph represents the price of the commodity in question, and the horizontal axis the quantity of the good. The graph provides a means of analysing the supply and demand at different prices. It is seen that (Fig. A.1), as the supply rises as the price of the product rises, the supply curve is upward sloping and, as the demand rises as the price of the product falls, the demand curve is downward sloping. The demand curve is effectively a summary of the various cost-benefit calculations that buyers make with respect to a good, or in another way reflects the set of price-quantity pairs for which buyers are satisfied.

The discussion so far has assumed that the supply and demand curves are fixed for the duration of market transactions. It must be noted that a change in demand is different from a change in the quantity demanded: a change in demand represents a shift in the entire demand curve, while a change in quantity demanded represents a movement along the demand curve. In economic terms the supply and demand curves in Figure A.1 depict the relation between price and quantity supplied (or demanded) *holding other things constant*. These 'other things' represent the factors that shape the supply or demand curve in the first place.

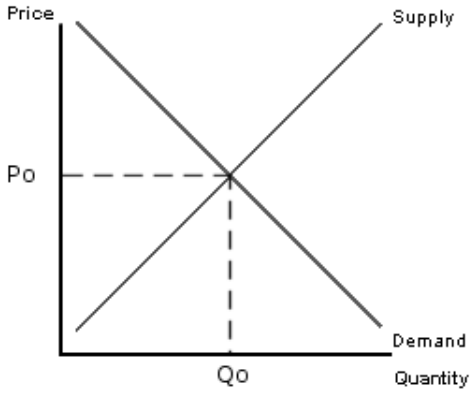

Figure A.1: Supply and Demand curves

For the supply curve these determining factors include:

- **Technology**: determines how much quantity of a commodity can be produced at a given price.
- **Input Costs**: includes factors such as labour and fuel costs which also affect how much of a commodity can be produced at a given price.
- **Government Regulation**: including, for example, safety regulation that puts a limit on the amount of commodity that can be produced at a given price.

The factors influencing the demand curve are summarised as:

- **The price of related goods**: the price of substitute goods affects the demand for the commodity in question.
- **Consumer incomes**: dictates the level of funds the buyers have to spend.
- **Tastes**: how 'popular' or sought after the commodity is (i.e. fashion, trends).

Of course these relate to external factors relevant to a real economy and not within an MBC system where the supply and demand curves are set arbitrarily before hand.

A.4 Experimental Economics

It is a fact that, after any reasonable length of time, human trading agents acting within a double-auction market do not have the cognitive power to make optimal decisions based on all of the transactions made in the market. And, as no-one can predict exactly what will happen in the future, it is said that humans do not act *rationally*. Instead, they act with a *bounded rationality*.

This makes the dynamics of human double-auctions somewhat unpredictable and so it is difficult to quantify to what extent the market behaviour is dependent on human intelligence or on the structure of the market itself. The field of *experimental economics* aimed to shed light on these issues by means of controlled experiments with human subjects.

Smith [15] carried out just such experiments and in each, every human subject was given the means to buy or sell a unit of commodity according to a limit price. This pre-determined the supply and demand in the (double-auction) market. From these tests a wide variety of adjustments could be made in order to study the dynamics of the market and how quickly the market approached equilibrium. In summary it was found that with a fairly small number of human traders the market could quickly converge on the equilibrium price.

Appendix B

Additional ZIP-Trading Agent Experimental Results

B.1 Introduction

This first part of this appendix presents further results obtained with the re-implementation of the ZIP-trading agent simulation using supply and demand schedules taken from [3]. The second part presents additional results evolving ZIP agent parameters, specifically an analysis of the parameter sets generated by the best evolutionary runs (as shown in Fig. 3.16) starting from 'zero' and 'hard' starting conditions.

B.2 Re-implementation of the ZIP-trading Agent Simulation

B.2.1 Flat Supply Curve

For this test the demand curve was set to remain the same as that shown in Fig. 3.1, but the supply curve set to be flat with all the seller agents' limit prices set at $2.00 (Fig B.1). This gives an equilibrium price of $2.00. The mean transaction price, alpha and profit dispersion plots generated from this test are shown in Figures B.2 through B.4. A sample transaction price time-series is shown in Figure B.5.

Figure B.2 shows the mean transaction price quickly falling into line with the $2.00 equilibrium price, and, as with the results given in [3], the approach to the equilibrium price happens from initially higher transaction prices (from above) rather than from initially lower prices. This indicates that the sellers have higher percentage profit margins than the buyers [3]. The alpha plot (Fig. B.3) indicates an even faster equilibration process than that shown in Figure 3.3, as does the profit dispersion (Fig. B.4), qualitively and quantitatively matching that shown in [3]. The sample transaction price series (Fig. B.5) also shows much less volatility than Figure 3.7 concerning early transaction prices.

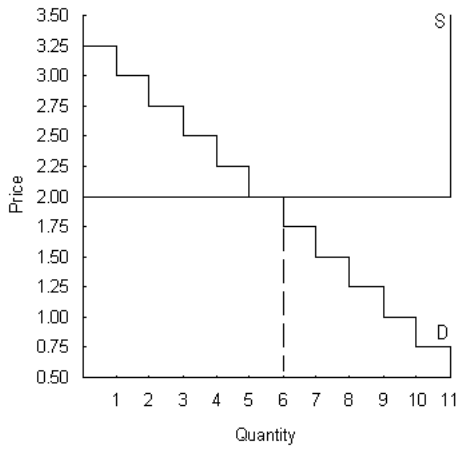

Figure B.1: Flat supply curve with 11 buyers and 11 sellers, where $P_0 = \$2.00$

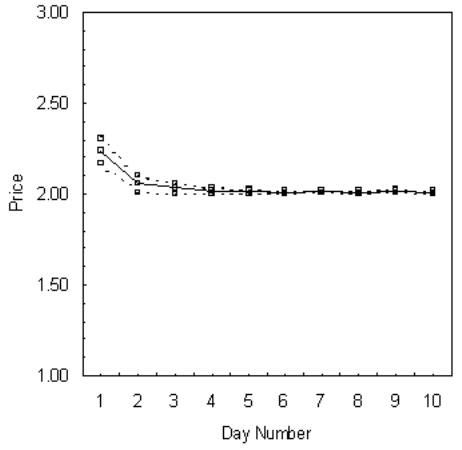

Figure B.2: Mean transaction prices, averaged over 50 ZIP trials, for the flat supply curve shown in Figure B.1 ($P_0 = \$2.00$). Format as for Fig. 3.2.

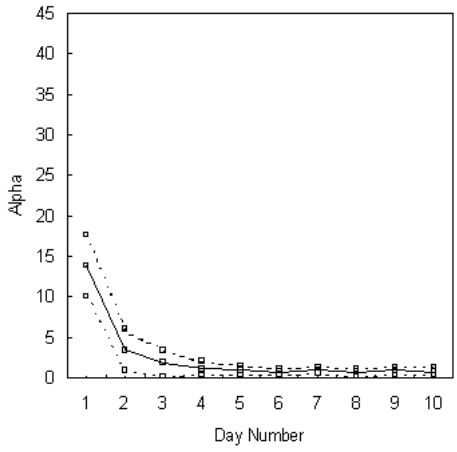

Figure B.3: Smith's alpha value, averaged over 50 ZIP trials, for the flat supply curve shown in Figure B.1 ($P_0 = \$2.00$). Format as for Fig. 3.2.

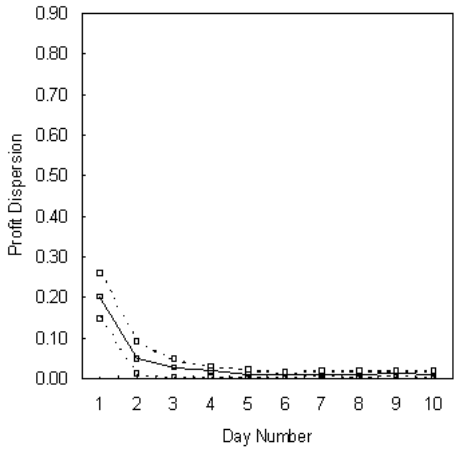

Figure B.4: Mean profit dispersion, averaged over 50 ZIP trials, for the flat supply curve shown in Figure B.1 ($P_0 = \$2.00$). Format as for Fig. 3.2.

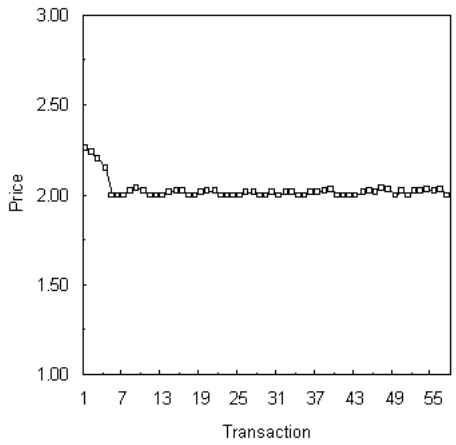

Figure B.5: Transaction-price time series from one 10-day market ZIP trial using the flat supply curve shown in Figure B.1 ($P_0 = \$2.00$).

B.2.2 Flat Curves with Excess Supply

For completeness, and in contrast to the test carried out in Section 3.3.2, the schedule for this test has an excess of supply (11 sellers and six buyers), and an equilibrium price of \$2.00 (Fig. B.6). The trading statistic plots for this test are shown in Figures B.7 to B.9, with a sample transaction price time-series taken from one of the individual trials in Figure B.10. The equilibration convergence indicated by these plots shows an improvement over the Section 3.3.2 test (and in terms of profit dispersion, as in [3]), and as expected this convergence happens from above in contrast to from below (as was the case where there was an excess demand).

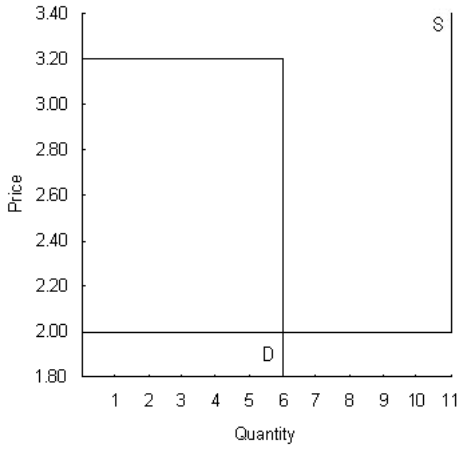

Figure B.6: Flat supply and demand curves with excess supply: 6 buyers and 11 sellers, where $P_0 = \$2.00$

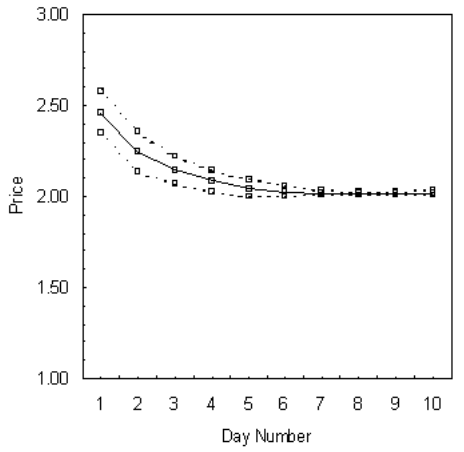

Figure B.7: Mean ZIP transaction prices, averaged over 50 ZIP trials, for the flat supply and demand curves shown in Figure B.6 ($P_0 = \$2.00$). Format as for Fig. 3.2.

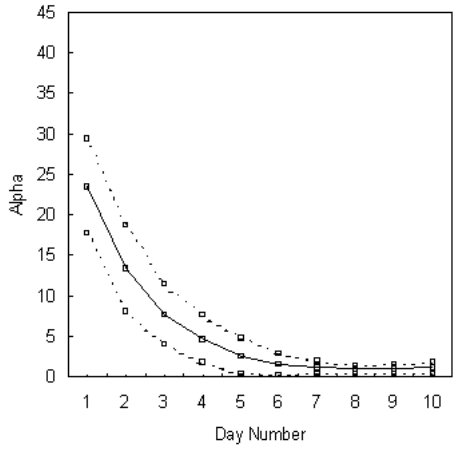

Figure B.8: Smith's alpha value, averaged over 50 ZIP trials, for the flat supply and demand curves shown in Figure B.6 (P_0 = \$2.00). Format as for Fig. 3.2.

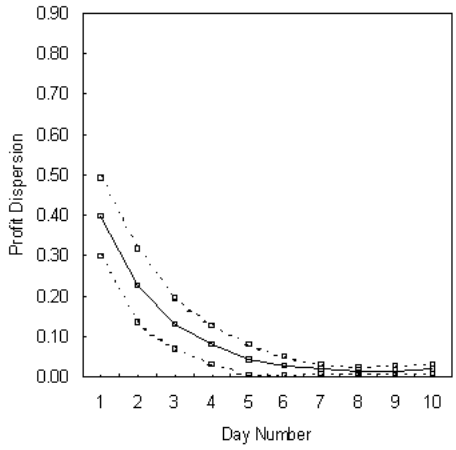

Figure B.9: Mean profit dispersion, averaged over 50 ZIP trials, for the flat supply and demand curves shown in Figure B.6 (P_0 = \$2.00). Format as for Fig. 3.2.

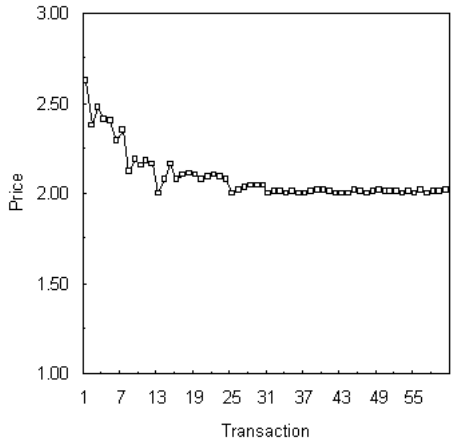

Figure B.10: Transaction-price time series from one 10-day market trial using the flat supply and demand curves shown in Figure B.6 (P_0 = \$2.00).

B.2.3 Shift in Demand Curve

For ZIP traders to be of any real use within an MBC system, it is imperative that they be able to adapt quickly and efficiently to (often constantly) changing market conditions. For example, allocating and re-organising resource usage equally well in times of peak demand as well as in times where units supplying the commodity in question fail (i.e. representing a fall in supply). This test demonstrates a simple shift in the demand of the market partway through a single trial in order to quantify whether this desired adaptation occurs.

Specifically, for the first 10 trading periods the ZIP trial initially uses the original supply and demand schedules shown in Figure 3.1, but then changes for an additional 5 trading periods to use a schedule where the demand curve has shifted upwards (Fig. B.11) – where \$0.50 is added to each buyer agents' limit price. The equilibrium price changes in this instance from \$2.00 to \$2.25, and it is expected that the ZIP agents will quickly adapt and converge to this new price. The trading statistic plots for this test are shown in Figures B.12 to B.14.

As expected, the mean transaction price plot (Fig. B.12) shows a rapid re-alignment towards the new equilibrium price, and this adaptation is confirmed

in the alpha (Fig. B.13) and profit dispersion (Fig. B.14) plots by the 'blip' starting on the 10th trading period as it takes the ZIP agents a couple of trading periods to complete the adjustment.

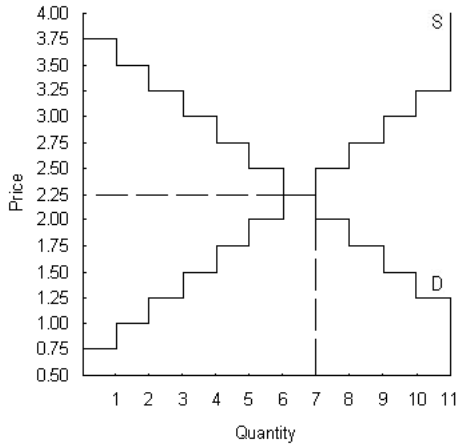

Figure B.11: Upward shifted demand curve on schedule shown in Figure 3.1, where $P_0 = \$2.25$.

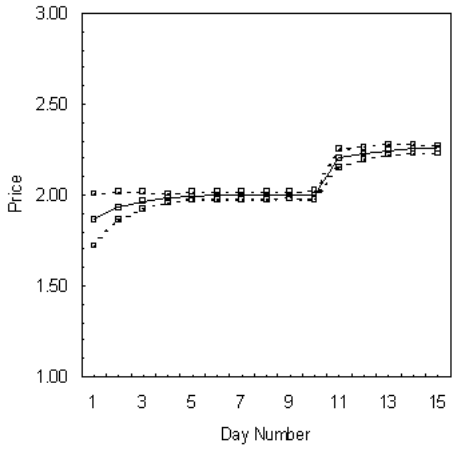

Figure B.12: Mean transaction prices, averaged over 50 ZIP trials, for a mid-trial change in supply and demand curves from that shown in Figure 3.1 to that shown in Figure B.11 (P_0 = \$2.25). Format as for Fig. 3.2.

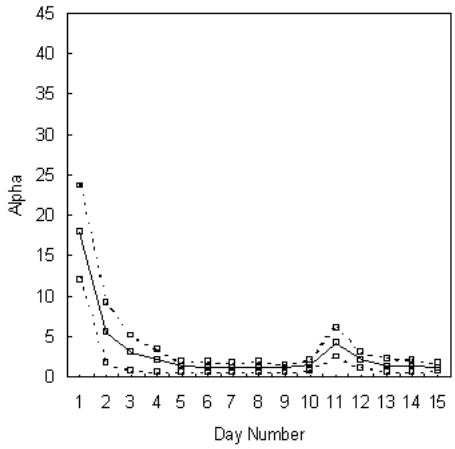

Figure B.13: Smith's alpha value, averaged over 50 ZIP trials, for a mid-trial change in supply and demand curves from that shown in Figure 3.1 to that shown in Figure B.11 (P_0 = \$2.25). Format as for Fig. 3.2.

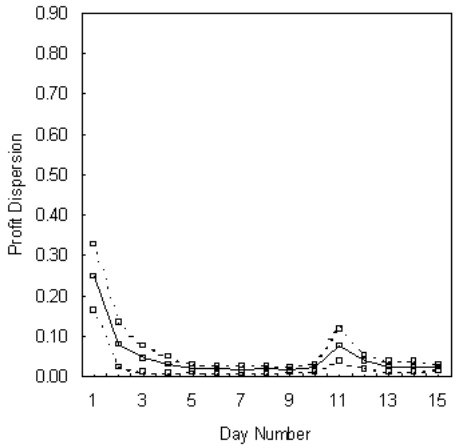

Figure B.14: Mean profit dispersion, averaged over 50 ZIP trials, for a mid-trial change in supply and demand curves from that shown in Figure 3.1 to that shown in Figure B.11 (P_0 = \$2.25). Format as for Fig. 3.2.

B.2.4 Symmetric Supply and Demand Curves using the NYSE Rule

Finally, use of the NYSE rule was made in order to see how it affected the dynamics of the market. The symmetric supply and demand curves, with an equilibrium price of \$2.00, were used (Fig. 3.1). Briefly, the NYSE acts as an improvement rule by stipulating that each bid or offer leading to a transaction must be an improvement on the previous bid or offer until a deal goes through, at which time the process starts again. The trading statistic plots for this test are given in Figures B.15 through B.17.

It is clear from the mean transaction price (Fig. B.15) and the sample transaction price time-series (Fig. B.17) plots that use of the NYSE rule is hampering the performance of the market as after the 10 trading periods of the trial, a full convergence to the equilibrium price of \$2.00 had not been made. This is due to the ZIP agents not getting access to and thus adapting themselves to bids or offers that are not necessarily an improvement over the previous quote price (have 'incomplete knowledge' [20]) – the NYSE rule restricts the full range of successful and unsuccessful quote prices from occurring in the market.

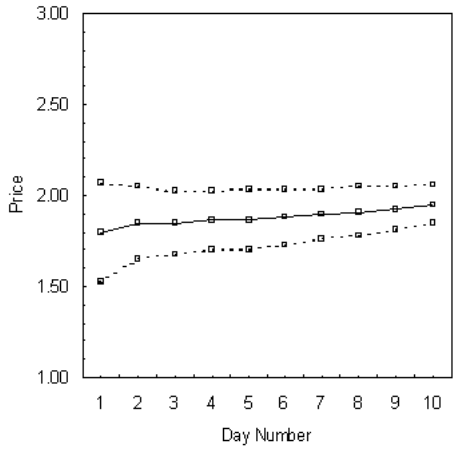

Figure B.15: Mean transaction prices, averaged over 50 ZIP trials using the NYSE rule, for the supply and demand curves in Figure 3.1 (P_0 = \$2.00). Format as for Fig. 3.2.

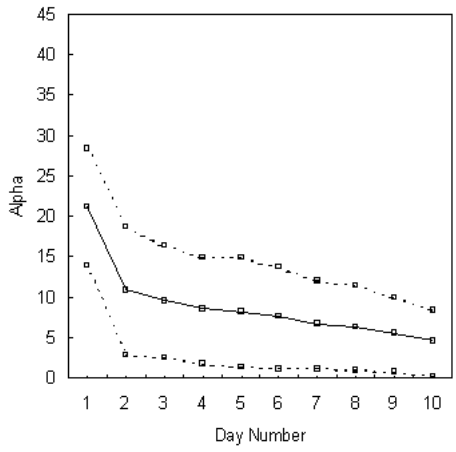

Figure B.16: Smith's alpha value, averaged over 50 ZIP trials using the NYSE rule, for the supply and demand curves in Figure 3.1 (P_0 = \$2.00). Format as for Fig. 3.2.

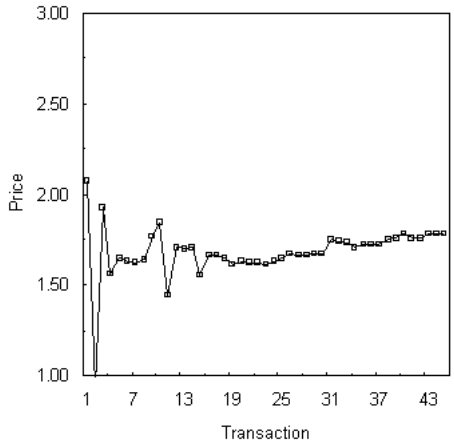

Figure B.17: Transaction-price time series from one 10-day market ZIP trial with the NYSE rule using the supply and demand curves in Figure 3.1 (P_0 = \$2.00).

B.3 Evolutionary Optimisation of ZIP-Trading Agent Parameter Sets

B.3.1 Zero Initial Conditions

Predictably, the sample transaction-price time series (Fig. B.18) and alpha plot (Fig. B.19) indicate that when setting the agent adaptation parameters to zero, the market is extremely volatile and no convergence is made to the equilibrium price (i.e. the agents cannot adapt). After the zero parameter values have evolved for 200 generations however, this situation improves considerably, as illustrated by the plots of Figures B.20 and B.21 showing an even better convergence to the equilibrium price than from the 'easy' initial conditions after 200 generations (Figures 3.19 and 3.20). The final generation elite genome for this run has the following parameter vector V_{zero}:

$$V_{zero} = [0.381, 0.194, 0.045, 0.005, 0.209, 0.090, 0.012, 0.063]$$

On average the parameter values in the vector matched fairly closely those found in the original results [16], and qualatively that of V_{easy}. The final recorded fitness value for this genome was 4.08, confirming the alpha (Fig. B.20)

and sample transaction price time-series (Fig. B.21) plots that indicated an even better performing parameter set than that found after 200 generations with the 'easy' initial conditions. However, because Figure 3.16 shows that the fitness values of these two evolutionary runs quickly reached to within the same fitness range this improved parameter vector cannot be a reflection on the initial 'zero' starting conditions.

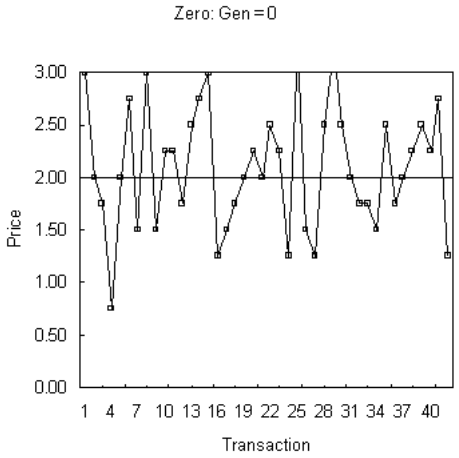

Figure B.18: Transaction-price time series from one trial of the elite genome in the first generation from 'zero' initial conditions.

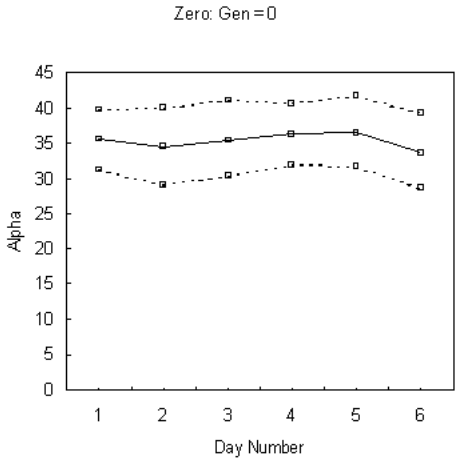

Figure B.19: Smith's alpha value, averaged over 50 ZIP trials, for the elite genome in the first generation from 'zero' initial conditions.

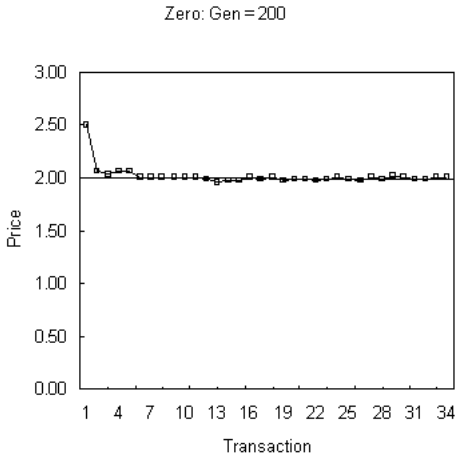

Figure B.20: Transaction-price time series from one trial of the elite genome in the final generation from 'zero' initial conditions.

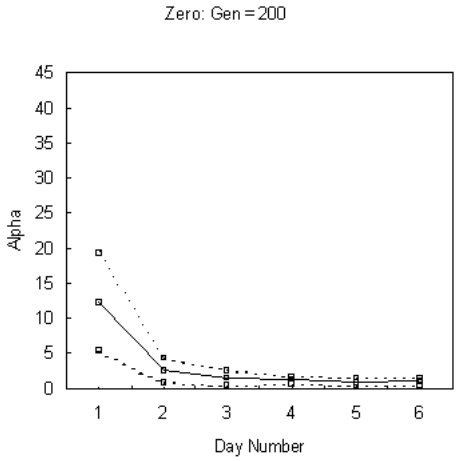

Figure B.21: Smith's alpha value, averaged over 50 ZIP trials, for the elite genome in the final generation from 'zero' initial conditions.

B.3.2 Hard Initial Conditions

As mentioned in Section 3.4.4, the 'hard' parameter values are highly sub-optimal and when used by the ZIP agents will, like the 'zero' parameters, produce erratic market dynamics with little or no convergence to the equilibrium price (Figures B.22 and B.23). However, although not desirable, the resulting market dynamics are not quite as bad as for 'zero', with the alpha values (Fig. B.23) for example being quantitatively lower. After the 'hard' parameters are evolved for 200 generations (Figures B.24 and B.25), the much more stable equilibrating behaviour as seen for the other 'best' evolutionary runs, results – in fact, their similarity to the final generation plots with 'zero' initial conditions means that they are not discussed further here. The final generation elite genome for this run has the following parameter vector V_{hard}:

$$V_{hard} = [0.473, 0.573, 0.213, 0.380, 0.200, 0.082, 0.012, 0.031]$$

While not quite as closely matching the final 'hard' parameter vector gained after 200 generations in the original experiments [16] (with a few of the parameters having a difference in value of 0.150 or more), a very favourable weighted alpha fitness score of 4.35 was attained, proving in no way that an inferior

genome was evolved in comparison to the 'easy' and 'zero' runs. This is perhaps not surprising given that the β_b, μ_b, μ_Δ, c_r and c_a (1st, 5th, 6th, 7th and 8th) parameters on the V_{hard} vector were quantitatively very similar to those on the V_{easy} and V_{zero} vectors. The bigger parameter value differences (of about 0.2 to 0.5) on the other parameters shows that a larger learning rate, momentum parameter and initial profit co-efficient can be favourable, and also that there is quite a large subset of parameter vectors that are able to give good performance when transplanted into the ZIP trading agents.

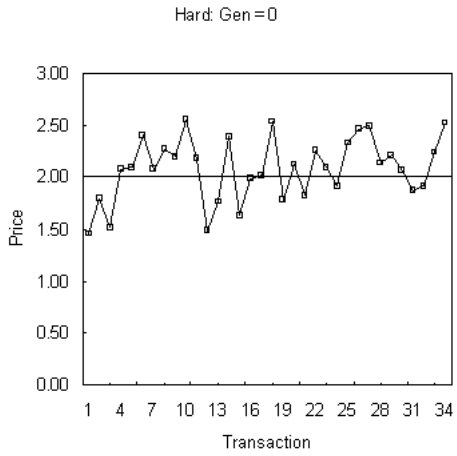

Figure B.22: Transaction-price time series from one trial of the elite genome in the first generation from 'hard' initial conditions.

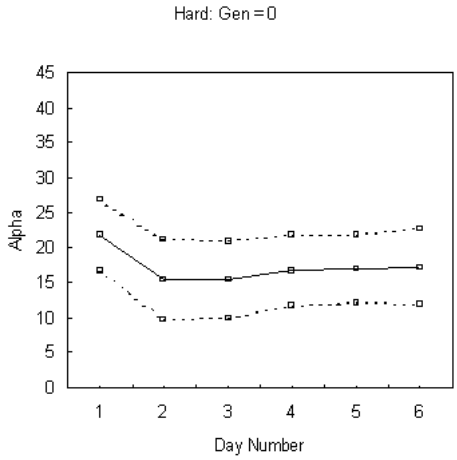

Figure B.23: Smith's alpha value, averaged over 50 ZIP trials, for the elite genome in the first generation from 'hard' initial conditions.

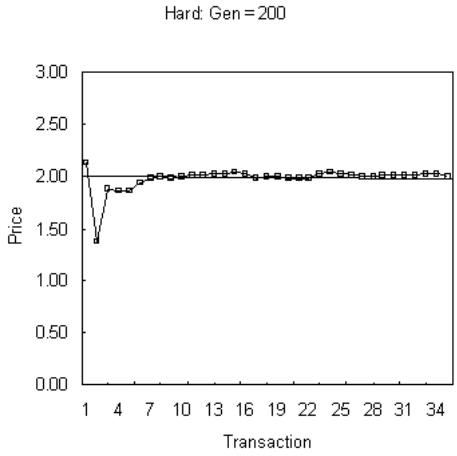

Figure B.24: Transaction-price time series from one trial of the elite genome in the final generation from 'hard' initial conditions.

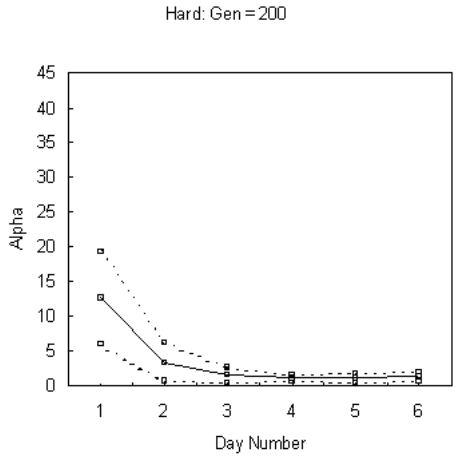

Figure B.25: Smith's alpha value, averaged over 50 ZIP trials, for the elite genome in the final generation from 'hard' initial conditions.

Appendix C

Source Code

C.1 ZIP Simulation Classes

The simulation source code is decomposed into a number of well-defined classes. A brief description accompanies each class, and each is comprehensively commented.

C.1.1 ZIP_Agent.java

Description: Defines a ZIP trading agent. Adapts the agents profit margin in response to successful and unsuccessful market quotes.

```
// Class:  ZIP_Agent.java
import java.io.PrintStream;
import java.util.Random;
public class ZIP_Agent implements ZIP_Constants
{
// ZIP_AGENT PARAMETERS
// agent buyer or seller
public boolean blnAGNTtype;
// agent being active in market or not
public boolean blnAGNTactive;
// if agent is willing to trade at this price
public boolean blnAGNTwilling;
// allowed to trade at this price?
public boolean blnAGNTable;
// number of deals done
public int intAGNTnumdeals;
// the bottom line price for this agent
public double dblAGNTlimit;
// profit coeff in determining bid/offer price (GA PARAMETER)
public double dblAGNTprofit;
// coeff for changing profit over time (beta) (GA PARAMETER)
public double dblAGNTlearnrate;
// momentum in changing profit (GA PARAMETER)
public double dblAGNTmomentum;
// last change
public double dblAGNTlastchange;
// what the agent will actually bid
public double dblAGNTprice;
// how much commodity the agent has
public double dblAGNTquantity;
// how much money the agent has in the bank
public double dblAGNTmoney;
// actual gain
public double dblAGNTactualgain;
// theoretical gain
public double dblAGNTtheorgain;
// in determining average reward
public double dblAGNTsum;
// average reward
public double dblAGNTaverage;
// agent adaptation parameters
public double dblAGNT_lr_L = 0.10;
public double dblAGNT_lr_H = 0.40;
public double dblAGNT_mm_L = 0.00;
public double dblAGNT_mm_H = 0.10;
public double dblAGNT_pf_L = 0.05;
public double dblAGNT_pf_H = 0.30;
public double dblAGNT_tr_R = 0.05;
public double dblAGNT_tr_A = 0.05;
```

```
// ---------------------------------------------------------------
// AGNT_CALCPRICE: set the price of an agent from its limit and profit
// values (called by ZIP_Sim.SIM_init_day, AGNT_profit_alter)
public void AGNT_calcPrice()
{
dblAGNTprice = dblAGNTlimit * ( 1 + dblAGNTprofit );
// normalise to 1.d.p
dblAGNTprice = ( Math.floor(( dblAGNTprice * 100 ) + 0.5 )) / 100;
}
// ---------------------------------------------------------------
// AGNT_INIT: intialise the common (buyer or seller) elements of an
// agent (called by ZIP_Agent.AGNT_init_buyer and AGNT_init_seller)
public void AGNT_init( boolean blnOutput, PrintStream psOUT, Random
random )
{
blnAGNTactive = true;
intAGNTnumdeals = 0;
// (random range [dblAGNT_lr_L, (dblAGNT_lr_L + dblAGNT_lr_H)])
dblAGNTlearnrate = dblAGNT_lr_L + ( random.nextDouble() *
dblAGNT_lr_H );
dblAGNTmoney = 0.0;
dblAGNTsum = 0.0;
dblAGNTlastchange = 0.0;
// (random range [dblAGNT_mm_L, (dblAGNT_mm_L + dblAGNT_mm_H)])
dblAGNTmomentum = dblAGNT_mm_L + ( random.nextDouble()
* dblAGNT_mm_H );
if ( blnOutput )
{
psOUT.println( " profit = " + dblAGNTprofit + ", beta = "
+ dblAGNTlearnrate + ", momentum = " + dblAGNTmomentum +
", money = " + dblAGNTmoney );
}
}
// ---------------------------------------------------------------
// AGNT_INIT_BUYER: initialise buyer agent specifics (called by
// ZIP_Sim.SIM_main )
public void AGNT_init_buyer( int intBuyer, boolean blnOutput,
PrintStream psOUT, Random random )
{
blnAGNTtype = BUY;
// initialise PROFIT VALUE (random range [dblAGNT_pf_L,
// (dblAGNT_pf_L + dblAGNT_pf_H)])
dblAGNTprofit = -1.0 * ( dblAGNT_pf_L + ( random.nextDouble()
* dblAGNT_pf_H ));
if ( blnOutput )
{
psOUT.print( "ZIP_AGENT: BUYER " + intBuyer );
}
// initialise common agent elements
AGNT_init( blnOutput, psOUT, random );
}
// ---------------------------------------------------------------
// AGNT_INIT_SELLER: initialise seller agent specifics (called by
// ZIP_Sim.SIM_main)
public void AGNT_init_seller( int intSeller, boolean blnOutput,
PrintStream psOUT, Random random )
{
blnAGNTtype = SELL;
// initialise PROFIT VALUE (random range [dblAGNT_pf_L,
// (dblAGNT_pf_L + dblAGNT_pf_H)])
dblAGNTprofit = dblAGNT_pf_L + ( random.nextDouble() *
dblAGNT_pf_H );
if ( blnOutput )
{
psOUT.print( "ZIP_AGENT: SELLER " + intSeller );
}
// initialise common agent elements
AGNT_init( blnOutput, psOUT, random );
```

```
}
// ---------------------------------------------------------------------
// AGNT_WILLING_TRADE: returns boolean indicating whether agent will
// trade at this price (called by ZIP_Sim.SIM_main,
// ZIP_Sim.SIM_get_willing)
public boolean AGNT_willing_trade( double dblPrice )
{
// if BUYER agent
if ( blnAGNTtype == BUY )
{
// if active and agents current bid price is >= to sellers offer
// price then set as willing
if (( blnAGNTactive ) && ( dblAGNTprice >= dblPrice ))
{
blnAGNTwilling = true;
}
else
{
blnAGNTwilling = false;
}
}
// if SELLER agent
else if ( blnAGNTtype == SELL )
{
// if active and agents current offer price is <= to buyers
// offer price then set as willing
if (( blnAGNTactive ) && ( dblAGNTprice <= dblPrice ))
{
blnAGNTwilling = true;
}
else
{
blnAGNTwilling = false;
}
}
return blnAGNTwilling;
}
// ---------------------------------------------------------------------
// AGNT_PROFIT_ALTER: update profit margin on basis of sale price
// using Widrow-Hoff style update with learning rate beta (called by
// ZIP_Agent.AGNT_shout_update_buyer and AGNT_shout_update_seller)
public void AGNT_profit_alter( int intAgent, boolean blnAgentType,
double dblPrice, boolean blnOutput, PrintStream psOUT,
int intStatus )
{
double dblPriceDiff;
double dblPriceChange;
double dblNewProfit;
String strAgentType = "";
if ( blnAgentType )
{
strAgentType = "BUYER";
}
else
{
strAgentType = "SELLER";
}
if ( blnOutput )
{
psOUT.println( "ZIP_AGENT: " + strAgentType + " " + intAgent
+ ": Limit = " + dblAGNTlimit + ", Profit = " +
dblAGNTprofit + ", Price " + dblPrice );
}
// calculate price difference between last marker quote price
// and agents (current) price
dblPriceDiff = dblPrice - dblAGNTprice;
// CALCULATE PRICE CHANGE using momentum in changing profit, price
// difference, last change etc.
dblPriceChange = ( ( 1.0 - dblAGNTmomentum ) * dblAGNTlearnrate
* dblPriceDiff ) + ( dblAGNTmomentum * dblAGNTlastchange );
```

```
if ( blnOutput )
{
psOUT.println( "ZIP_AGENT: " + strAgentType + " " + intAgent +
": Last Change = " + dblAGNTlastchange +
", Price Difference = " + dblPriceDiff +
", Price Change = " + dblPriceChange );
}
// make the price change the last change value for agent
dblAGNTlastchange = dblPriceChange;
// set new prices by altering profit margin (keep within bottom
// line price for agent)
dblNewProfit = (( dblAGNTprice + dblPriceChange ) / dblAGNTlimit )
 - 1.0;
// (only going through here if there is a deal...)
if ( blnAGNTtype == SELL )
{
if ( dblNewProfit > 0.0 )
{
dblAGNTprofit = dblNewProfit;
}
}
else
{
if ( dblNewProfit < 0.0 )
{
dblAGNTprofit = dblNewProfit;
}
}
// set the price of an agent from its limit and profit values
AGNT_calcPrice();
if ( blnOutput )
{
psOUT.println( "ZIP_AGENT: " + strAgentType + " " + intAgent +
": New Profit = " + dblAGNTprofit + ", New Price = "
+ dblAGNTprice );
psOUT.println();
}
}
// ---------------------------------------------------------------
// AGNT_SHOUT_UPDATE_BUYER: update strategy of buyer agent after a
// shout (called by ZIP_Sim.SIM_main, ZIP_Sim.SIM_trade)
public void AGNT_shout_update_buyer( int intBuyer,
boolean blnDealType, int intStatus, double dblPrice,
boolean blnOutput, PrintStream psOUT, Random random )
{
boolean blnAgentType = true;
double dblTargetPrice = 0.0;
if ( blnOutput )
{
psOUT.println( "ZIP_AGENT: BUYER " + intBuyer + ": Active = "
+ blnAGNTactive );
}
// IF A DEAL has taken place
if ( intStatus == DEAL )
{
// if agents current bid price is >= deal price (bid price too
// high)
if ( dblAGNTprice >= dblPrice )
{
// see if can get lower price by cutting bid price (hence
// raise profit margin)
dblTargetPrice = ( dblPrice * ( 1.0 - ( random.nextDouble()
* dblAGNT_tr_R ))) - ( random.nextDouble()
* dblAGNT_tr_A );
// (update profit margin on basis of sale price using
// Widrow-Hoff update with learning rate)
AGNT_profit_alter( intBuyer, blnAgentType, dblTargetPrice,
blnOutput, psOUT, intStatus );
}
```

```
else
{
// bid price too low so wouldn't have got this deal...
if (( blnDealType == OFFER ) &&
( AGNT_willing_trade( dblPrice ) == false ) &&
( blnAGNTactive ))
{
// ...so raise bid price (hence reducing profit)
dblTargetPrice = ( dblPrice * ( 1.0 + (
random.nextDouble() * dblAGNT_tr_R ))) +
( random.nextDouble() * dblAGNT_tr_A );
// (update profit margin on basis of sale price using
// Widrow-Hoff update with learning rate)
AGNT_profit_alter( intBuyer, blnAgentType, dblTargetPrice,
 blnOutput, psOUT, intStatus );
}
}
}
// IF NO DEAL taken place
else
{
if ( blnDealType == BID )
{
// if agents current bid price is <= deal price (bid price
// too low)
if (( dblAGNTprice <= dblPrice ) && ( blnAGNTactive ))
{
// would have bid less and also lost the deal, so raise
// bid price (reduce profit)
dblTargetPrice = ( dblPrice * ( 1.0 + (
random.nextDouble() * dblAGNT_tr_R ))) +
( random.nextDouble() * dblAGNT_tr_A );
// (update profit margin on basis of sale price using
// Widrow-Hoff update with learning rate)
AGNT_profit_alter( intBuyer, blnAgentType, dblTargetPrice,
blnOutput, psOUT, intStatus );
}
}
}
}
// ---------------------------------------------------------------------
// AGNT_SHOUT_UPDATE_SELLER: update strategy of seller agent after a
// shout (called by ZIP_Sim.SIM_main, ZIP_Sim.SIM_trade)
public void AGNT_shout_update_seller( int intSeller,
boolean blnDealType, int intStatus, double dblPrice,
boolean blnOutput, PrintStream psOUT, Random random )
{
double dblTargetPrice;
boolean blnAgentType = false;
if ( blnOutput )
{
psOUT.println( "ZIP_AGENT: SELLER " + intSeller + ": Active = "
+ blnAGNTactive );
}
// IF A DEAL has taken place
if ( intStatus == DEAL )
{
// if agents current offer price <= deal price (offer too low)
if ( dblAGNTprice <= dblPrice )
{
// see if can get more by raising offer price (try and
// increase profits next time around)
dblTargetPrice = ( dblPrice * ( 1.0 + ( random.nextDouble()
* dblAGNT_tr_R ))) + ( random.nextDouble()
* dblAGNT_tr_A );
AGNT_profit_alter( intSeller, blnAgentType, dblTargetPrice,
blnOutput, psOUT, intStatus );
}
else
{
```

```
// offer too high so wouldn't have got this deal...
if (( blnDealType == BID ) &&
( AGNT_willing_trade( dblPrice ) == false ) &&
( blnAGNTactive == true ))
{
// ...so reduce offer price (decrease profit margin)
dblTargetPrice = ( dblPrice * ( 1.0 - (
random.nextDouble() * dblAGNT_tr_R ))) -
( random.nextDouble() * dblAGNT_tr_A );
AGNT_profit_alter( intSeller, blnAgentType, dblTargetPrice,
blnOutput, psOUT, intStatus );
}
}
}
// IF NO DEAL taken place
else
{
if ( blnDealType == OFFER )
{
// if agents current bid price is >= deal price (offer too
// high)
if (( dblAGNTprice >= dblPrice ) && ( blnAGNTactive ))
{
// would have asked for more and lost the deal, so
// reduce profit
dblTargetPrice = ( dblPrice * ( 1.0 - (
random.nextDouble() * dblAGNT_tr_R ))) - (
random.nextDouble() * dblAGNT_tr_A );
AGNT_profit_alter( intSeller, blnAgentType,
dblTargetPrice, blnOutput, psOUT, intStatus );
}
}
}
}
// ---------------------------------------------------------------------
}
```

C.1.2 ZIP_Constants.java

Description: System wide constants.

```
// Class:  ZIP_Exp_Control.java
public interface ZIP_Constants
{
// SYSTEM WIDE CONSTANTS
public static int MAX_N_DAYS = 30;
public static int MAX_TRADES = 100;
public static int TOT_TRADES = ( MAX_N_DAYS * MAX_TRADES );
public static int MAX_BUYERS = 100;
public static int MAX_SELLERS = 100;
public static int MAX_AGENTS = 100;
// max no. of bid/offer fails before day's trading closes
public static int MAX_FAILS = 100;
// max no. of units an agent can sell / buy
public static int MAX_UNITS = 3;
// max no. of supply or demand schedules in an experiment
public static int MAX_SCHED = 2;
public static int NULL_EQ = -1;
public static int EQ_THEORY = 0;
public static int EQ_ACTUAL = 1;
// AGENT CONSTANTS
// agent type
public static boolean BUY = true;
// agent type
public static boolean SELL = false;
// deal/shout type
public static boolean BID = true;
// deal/shout type
public static boolean OFFER = false;
// status (shout accepted)
public static int DEAL = 1;
// status (shout rejected)
public static int NO_DEAL = 0;
// status (end day)
public static int END_DAY = 2;
}
```

C.1.3 ZIP_Data_Day.java with GA Fitness Evaluation

Description: Records system day statistics, and for the GA calculates the weighted alpha fitness score.

```
// Class:  ZIP_Data_Day.java
import java.io.PrintStream;
public class ZIP_Data_Day implements ZIP_Constants
{
// SMITH'S ALPHA
ZIP_Day_Stat dblSTAT_alpha;
// QUANTITY
ZIP_Day_Stat dblSTAT_quant;
// EFFICIENCY
ZIP_Day_Stat dblSTAT_effic;
// PRICE
ZIP_Day_Stat dblSTAT_price;
// PROFIT DISPERSAL
ZIP_Day_Stat dblSTAT_pdisp;
// TRANSACTION PRICE VOLATILITY
ZIP_Day_Stat dblSTAT_volty;
// (used to dodge rounding errors on sqrt)
public static double SMALLREAL = 0.0000001;
public static int DD_ALPHA = 0;
public static int DD_QUANT = 1;
public static int DD_EFFIC = 2;
public static int DD_PRICE = 3;
public static int DD_PDISP = 4;
public static int DD_VOLTY = 5;
// ------------------------------------------------------------------
// STAT_ZERO: set everything to zero in one Real-stat structure
// (called by ZIP_Data_Day.data_day_init)
public void stat_zero( ZIP_Day_Stat day_stat )
{
// initialise
day_stat = new ZIP_Day_Stat();
day_stat.dblSum = 0.0;
day_stat.dblSumSq = 0.0;
day_stat.intN = 0;
}
// ------------------------------------------------------------------
// DATA_DAY_INIT: initialise day data (called by ZIP_Sim.SIM_main)
public void data_day_init()
{
dblSTAT_alpha = new ZIP_Day_Stat();
dblSTAT_quant = new ZIP_Day_Stat();
dblSTAT_effic = new ZIP_Day_Stat();
dblSTAT_price = new ZIP_Day_Stat();
dblSTAT_pdisp = new ZIP_Day_Stat();
dblSTAT_volty = new ZIP_Day_Stat();
stat_zero( dblSTAT_alpha );
stat_zero( dblSTAT_quant );
stat_zero( dblSTAT_effic );
stat_zero( dblSTAT_price );
stat_zero( dblSTAT_pdisp );
stat_zero( dblSTAT_volty );
}
// ------------------------------------------------------------------
// DATA_DAY_UPDATE: update day data (called by ZIP_Sim.SIM_main)
public void data_day_update( int intNoDeals, double dblSumPrice,
double dblAlpha, double dblPDisp, double dblEffic,
double dblPriceDiff )
{
if ( intNoDeals > 0 )
{
```

```
dblSTAT_price.dblSum += ( dblSumPrice / intNoDeals );
dblSTAT_price.dblSumSq += (( dblSumPrice / intNoDeals ) *
( dblSumPrice / intNoDeals ));
dblSTAT_price.intN++;
double TMPdblVolty = Math.sqrt( dblPriceDiff / intNoDeals );
dblSTAT_volty.dblSum += TMPdblVolty;
dblSTAT_volty.dblSumSq += ( TMPdblVolty * TMPdblVolty );
dblSTAT_volty.intN++;
dblSTAT_alpha.dblSum += dblAlpha;
dblSTAT_alpha.dblSumSq += ( dblAlpha * dblAlpha );
dblSTAT_alpha.intN++;
dblSTAT_effic.dblSum += dblEffic;
dblSTAT_effic.dblSumSq += ( dblEffic * dblEffic );
dblSTAT_effic.intN++;
dblSTAT_quant.dblSum += intNoDeals;
dblSTAT_quant.dblSumSq += ( intNoDeals * intNoDeals );
dblSTAT_quant.intN++;
dblSTAT_pdisp.dblSum += dblPDisp;
dblSTAT_pdisp.dblSumSq += ( dblPDisp * dblPDisp );
dblSTAT_pdisp.intN++;
}
}
// ------------------------------------------------------------------
// DATA_DAY_STATS: output mean plus and minus one standard deviation
// (called by ZIP_Sim.SIM_main)
public void data_day_stats( String strField, double[] TMPdblSum,
double[] TMPdblSumSq, int[] TMPintN, int intNoDays, int intExpNo,
double[] dbl_Mean, double[] dbl_m1SD, double[] dbl_p1SD,
PrintStream psDAY )
{
psDAY.println( " " + strField + " (MEAN) n=" + intExpNo );
for ( int intDayNo = 0; intDayNo < intNoDays; intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " " + ( TMPdblSum[intDayNo]
/ TMPintN[intDayNo] ) );
dbl_Mean[intDayNo] += ( TMPdblSum[intDayNo] /
TMPintN[intDayNo] );
}
if ( intExpNo > 0 )
{
psDAY.println( " " + strField + " (-S.D.) n=" + intExpNo );
for ( int intDayNo = 0; intDayNo < intNoDays; intDayNo++ )
{
double TMPdblMean = TMPdblSum[intDayNo] / TMPintN[intDayNo];
double TMPdblMeanSq = TMPdblMean * TMPdblMean;
double TMPdblDiff = ( TMPdblSumSq[intDayNo] /
TMPintN[intDayNo] ) - TMPdblMeanSq;
if ( TMPdblDiff < SMALLREAL )
{
TMPdblDiff = 0.0;
}
psDAY.println( ( intDayNo + 1 ) + " " + ( TMPdblMean -
Math.sqrt( TMPdblDiff )));
dbl_m1SD[intDayNo] += ( TMPdblMean -
Math.sqrt( TMPdblDiff ));
}
psDAY.println( " " + strField + " (+S.D.) n=" + intExpNo );
for ( int intDayNo = 0; intDayNo < intNoDays; intDayNo++ )
{
double TMPdblMean = TMPdblSum[intDayNo] / TMPintN[intDayNo];
double TMPdblMeanSq = TMPdblMean * TMPdblMean;
double TMPdblDiff = ( TMPdblSumSq[intDayNo] /
TMPintN[intDayNo] ) - TMPdblMeanSq;
if ( TMPdblDiff < SMALLREAL )
{
TMPdblDiff = 0.0;
}
psDAY.println( ( intDayNo + 1 ) + " " + ( TMPdblMean +
```

```
Math.sqrt( TMPdblDiff ));
dbl_p1SD[intDayNo] += ( TMPdblMean +
Math.sqrt( TMPdblDiff ));
}
}
}
// ---------------------------------------------------------------------
// DATA_DAY_ALPHA_FITNESS: used to calculate GA fitness value which is
// a weighted n-day combination of mean alpha parameter values.
// (called by ZIP_Sim.SIM_main)
public double data_day_alpha_fitness( double[] TMPdblSum,
int[] TMPintN )
{
// day 1 has a weighting of 1.75
double dblALPHA_day1 = ( TMPdblSum[0] / TMPintN[0] ) * 1.75;
// day 2 has a weighting of 1.5
double dblALPHA_day2 = ( TMPdblSum[1] / TMPintN[1] ) * 1.5;
// day 3 has a weighting of 1.25
double dblALPHA_day3 = ( TMPdblSum[2] / TMPintN[2] ) * 1.25;
// days 4, 5 and 6 have a weighting of 1.0
double dblALPHA_day4 = TMPdblSum[3] / TMPintN[3];
double dblALPHA_day5 = TMPdblSum[4] / TMPintN[4];
double dblALPHA_day6 = TMPdblSum[5] / TMPintN[5];
double dblExpFitness = ( dblALPHA_day1 + dblALPHA_day2
+ dblALPHA_day3 + dblALPHA_day4 + dblALPHA_day5
+ dblALPHA_day6 ) / 7.5;
return dblExpFitness;
}
// ---------------------------------------------------------------------
}
// data type for stat sum and sum of squares, used in calculating mean
// and s.d. (used by ZIP_Data_Day)
class ZIP_Day_Stat implements ZIP_Constants
{
public int intN;
public double dblSum;
public double dblSumSq;
}
// ---------------------------------------------------------------------
```

C.1.4 ZIP_Data_Trade.java

Description: Defines a structure for recording the details of a single trade: the transaction price, whether the quote was a bid or an offer, and what the theoretical and actual equilibrium prices were at the time of the trade.

```
// Class:  ZIP_Data_Trade.java
public class ZIP_Data_Trade implements ZIP_Constants
{
// price at which deal succeeds
public double dblTDATdeal_price;
// type of deal accepted (bid or offer)
public boolean blnTDATdeal_type;
// theoretical equilibrium price
public double dblTDATtheor_eq_price;
// theoretical equilibrium quantity
public int intTDATtheor_eq_quant;
// actual equilibrium price
public double dblTDATactual_eq_price;
// actual equilibrium quantity
public int intTDATactual_eq_quant;
}
```

C.1.5 ZIP_Exp_Control.java

Description: Reads in experiment control parameters and supply / demand schedule data.

```
// Class:  ZIP_Exp_Control.java
import java.io.PrintStream;
public class ZIP_Exp_Control implements ZIP_Constants
{
// ZIP_Exp_Control PARAMETERS
// id characters for output files
public String strEXPCid;
// number of trading periods to run for
public int intEXPCno_days;
// minimum number of trades per day
public int intEXPCmin_trades;
// maximum number of trades per day
public int intEXPCmax_trades;
// 0 = ZIP, 1 = ZI-C
public boolean blnEXPCrandom;
// 0 = nyse off, 1 = nyse on
public boolean blnEXPCnyse;
// number of demand schedules
public int intEXPCno_ds;
// number of supply schedules
public int intEXPCno_ss;
// details of ds's
public ZIP_Sched_SD[] EXPCds = new ZIP_Sched_SD[MAX_SCHED];
// details of ss's
public ZIP_Sched_SD[] EXPCss = new ZIP_Sched_SD[MAX_SCHED];
// index of currently active ds
public int intEXPCds;
// index of currently active ss
public int intEXPCss;
ZIP_Exp_Params expParams;
// ---------------------------------------------------------------------
// EXPC_IN: get control file data (called by: ZIP_Sim.SIM_main)
public void EXPC_in( boolean blnOutput, PrintStream psOUT )
{
expParams = new ZIP_Exp_Params();
// get strEXPCid
strEXPCid = expParams.strEXPCid;
// get intEXPCno_days
intEXPCno_days = expParams.intEXPCno_days;
if (( intEXPCno_days < 1 ) || ( intEXPCno_days > MAX_N_DAYS ))
{
psOUT.println( "FAIL: # trading days must be in range {1,...,"
+ MAX_N_DAYS + "}" );
System.out.println(
"FAIL: # trading days must be in range {1,...,"
+ MAX_N_DAYS + "}" );
System.exit( 0 );
}
// get intEXPCmin_trades
intEXPCmin_trades = expParams.intEXPCmin_trades;
if (( intEXPCmin_trades < 1 ) ||
( intEXPCmin_trades > MAX_TRADES ))
{
psOUT.println( "FAIL: min # trades must be in range {1,...,"
+ MAX_TRADES + "}" );
System.out.println(
"FAIL: min # trades must be in range {1,...,"
+ MAX_TRADES + "}" );
System.exit( 0 );
}
// get intEXPCmax_trades
```

```
intEXPCmax_trades = expParams.intEXPCmax_trades;
if (( intEXPCmax_trades < intEXPCmin_trades ) ||
( intEXPCmax_trades > MAX_TRADES ))
{
psOUT.println( "FAIL: max # trades must be in range {"
+ intEXPCmin_trades + ",...," + MAX_TRADES + "}" );
System.out.println( "FAIL: max # trades must be in range {"
+ intEXPCmin_trades + ",...," + MAX_TRADES + "}" );
System.exit( 0 );
}
// get blnEXPCrandom
blnEXPCrandom = expParams.blnEXPCrandom;
// get blnEXPCnyse
blnEXPCnyse = expParams.blnEXPCnyse;
// get intEXPCno_ds
intEXPCno_ds = expParams.intEXPCno_ds;
if (( intEXPCno_ds < 1 ) || ( intEXPCno_ds > MAX_SCHED ))
{
psOUT.println(
"FAIL: # demand schedules must be in range {1,...,"
+ MAX_SCHED + "}" );
System.out.println(
"FAIL: # demand schedules must be in range {1,...,"
+ MAX_SCHED + "}" );
System.exit( 0 );
}
// get intEXPCno_ss
intEXPCno_ss = expParams.intEXPCno_ss;
if (( intEXPCno_ss < 1 ) || ( intEXPCno_ss > MAX_SCHED ))
{
psOUT.println(
"FAIL: # supply schedules must be in range {1,...,"
+ MAX_SCHED + "}" );
System.out.println(
"FAIL: # supply schedules must be in range {1,...,"
+ MAX_SCHED + "}" );
System.exit( 0 );
}
// output control file parameter values to the log file
if ( blnOutput )
{
psOUT.println();
psOUT.println( "ZIP_EXP_CONTROL: ID string: " + strEXPCid );
psOUT.println( "ZIP_EXP_CONTROL: Number of days: "
+ intEXPCno_days );
psOUT.println(
"ZIP_EXP_CONTROL: Minimum number of trades per day: "
+ intEXPCmin_trades );
psOUT.println(
"ZIP_EXP_CONTROL: Maximum number of trades per day: "
+ intEXPCmax_trades );
if ( blnEXPCrandom == true )
{
psOUT.println( "ZIP_EXP_CONTROL: Random (ZI-C) traders" );
}
else
{
psOUT.println(
"ZIP_EXP_CONTROL: Intelligent (ZIP) traders" );
}
if ( blnEXPCnyse == true )
{
psOUT.println( "ZIP_EXP_CONTROL: NYSE trading rules" );
}
else
{
psOUT.println( "ZIP_EXP_CONTROL: No NYSE trading rules" );
}
psOUT.println( "ZIP_EXP_CONTROL: Number of demand schedules: "
```

```
+ intEXPCno_ds );
psOUT.println( "ZIP_EXP_CONTROL: Number of supply schedules: "
+ intEXPCno_ss );
psOUT.println();
}
// get demand schedules
for ( int intSched = 0; intSched < intEXPCno_ds; intSched++ )
{
if ( blnOutput )
{
psOUT.println( "ZIP_EXP_CONTROL: Demand schedule "
+ intSched );
}
// initialise array
EXPCds[intSched] = new ZIP_Sched_SD();
ZIP_Sched_SD s = expParams.EXPCds[intSched];
EXPC_read_schedule( EXPCds[intSched], s, blnOutput, psOUT );
}
// set index of currently active demand schedule to 0
intEXPCds = 0;
EXPCds[intEXPCds].intSDSHfirst_day = 0;
// get supply schedules
for ( int intSched = 0; intSched < intEXPCno_ss; intSched++ )
{
if ( blnOutput )
{
psOUT.println( "ZIP_EXP_CONTROL: Supply schedule "
+ intSched );
}
// initialise array
EXPCss[intSched] = new ZIP_Sched_SD();
ZIP_Sched_SD s = expParams.EXPCss[intSched];
EXPC_read_schedule( EXPCss[intSched], s, blnOutput, psOUT );
}
// set index of currently active supply schedule to 0
intEXPCss = 0;
EXPCds[intEXPCss].intSDSHfirst_day = 0;
}
// ---------------------------------------------------------------------
// EXPC_READ_SCHEDULE: read supply or demand schedule (called by
// ZIP_Exp_Control.EXPC_in)
public void EXPC_read_schedule( ZIP_Sched_SD schedule, ZIP_Sched_SD s,
boolean blnOutput, PrintStream psOUT )
{
// get intSDSHno_agents
schedule.intSDSHno_agents = s.intSDSHno_agents;
if (( schedule.intSDSHno_agents < 1 ) ||
( schedule.intSDSHno_agents > MAX_AGENTS ))
{
psOUT.println( "FAIL: # agents must be in range {1,...,"
+ MAX_AGENTS + "}" );
System.out.println( "FAIL: # agents must be in range {1,...,"
+ MAX_AGENTS + "}" );
System.exit( 0 );
}
// get intSDSHfirst_day
schedule.intSDSHfirst_day = s.intSDSHfirst_day;
// get intSDSHlast_day
schedule.intSDSHlast_day = s.intSDSHlast_day;
if ( schedule.intSDSHlast_day < schedule.intSDSHfirst_day )
{
psOUT.println( "FAIL: Last Day " + schedule.intSDSHlast_day
+ " < First Day " + schedule.intSDSHfirst_day );
System.out.println(
"FAIL: Last Day " + schedule.intSDSHlast_day
+ " < First Day " + schedule.intSDSHfirst_day );
System.exit( 0 );
```

```
}
// get blnSDSHcan_shout
schedule.blnSDSHcan_shout = s.blnSDSHcan_shout;
// output control file parameter values to the log file
if ( blnOutput )
{
psOUT.println( "ZIP_EXP_CONTROL: No. of agents: "
+ schedule.intSDSHno_agents );
psOUT.println( "ZIP_EXP_CONTROL: First day: "
+ schedule.intSDSHfirst_day );
psOUT.println( "ZIP_EXP_CONTROL: Last day: "
+ schedule.intSDSHlast_day );
if ( schedule.blnSDSHcan_shout )
{
psOUT.println( "ZIP_EXP_CONTROL: Traders can shout" );
}
else
{
psOUT.println( "ZIP_EXP_CONTROL: Traders cannot shout" );
}
psOUT.println();
}
// get intAGSHno_units
for ( int intAgent = 0; intAgent < schedule.intSDSHno_agents;
intAgent++ )
{
// initialise array
schedule.SDSHagent[intAgent] = new ZIP_Sched_Agent();
schedule.SDSHagent[intAgent].intAGSHno_units =
s.SDSHagent[intAgent].intAGSHno_units;
if (( schedule.SDSHagent[intAgent].intAGSHno_units < 1 ) ||
( schedule.SDSHagent[intAgent].intAGSHno_units >
MAX_UNITS ))
{
psOUT.println( "FAIL: # units must be in range {1,..., "
+ MAX_UNITS + "} " );
System.out.println( "FAIL: # units must be in range {1,..., "
+ MAX_UNITS + "} " );
System.exit( 0 );
}
if ( blnOutput )
{
psOUT.println( "ZIP_EXP_CONTROL: Agent " + intAgent +
" number of units: "
+ schedule.SDSHagent[intAgent].intAGSHno_units );
}
// get dblAGSHlimit (loop through units)
for ( int intUnit = 0; intUnit <
schedule.SDSHagent[intAgent].intAGSHno_units;
intUnit++ )
{
schedule.SDSHagent[intAgent].dblAGSHlimit[intUnit] =
s.SDSHagent[intAgent].dblAGSHlimit[intUnit];
if ( schedule.SDSHagent[intAgent].dblAGSHlimit[intUnit]
< 0.0 )
{
psOUT.println( "FAIL: negative price " +
s.SDSHagent[intAgent].dblAGSHlimit[intUnit] );
System.out.println( "FAIL: negative price " +
s.SDSHagent[intAgent].dblAGSHlimit[intUnit] );
System.exit( 0 );
}
if ( blnOutput )
{
psOUT.println( "ZIP_EXP_CONTROL: Limit price for unit "
+ intUnit + ": "
+ schedule.SDSHagent[intAgent].dblAGSHlimit[intUnit] );
}
```

```
}
}
if ( blnOutput )
{
psOUT.println();
}
}
// ---------------------------------------------------------------------
}
```

C.1.6 ZIP_Exp_Params.java

Description: The ZIP experimental parameters.

```
// Class:  ZIP_Exp_Params.java
// (used by ZIP_Exp_Control)
public class ZIP_Exp_Params
{
// ZIP_Control_File PARAMETERS
public String strEXPCid;
public int intEXPCno_days;
public int intEXPCmin_trades;
public int intEXPCmax_trades;
public boolean blnEXPCrandom;
public boolean blnEXPCnyse;
public int intEXPCno_ds;
public int intEXPCno_ss;
public ZIP_Sched_SD[] EXPCds;
public ZIP_Sched_SD[] EXPCss;
public ZIP_Exp_Params()
{
strEXPCid = "zip1hi";
intEXPCno_days = 10;
intEXPCmin_trades = 9;
intEXPCmax_trades = 9;
blnEXPCrandom = false;
blnEXPCnyse = false;
intEXPCno_ds = 1;
intEXPCno_ss = 1;
EXPCds = new ZIP_Sched_SD[intEXPCno_ds];
EXPCss = new ZIP_Sched_SD[intEXPCno_ss];
for ( int intSchedule = 0; intSchedule < intEXPCno_ds; intSchedule++ )
{
EXPCds[intSchedule] = new ZIP_Sched_SD();
}
for ( int intSchedule = 0; intSchedule < intEXPCno_ss; intSchedule++ )
{
EXPCss[intSchedule] = new ZIP_Sched_SD();
}
// set values for all demand schedule parameters
EXPCds[0].intSDSHno_agents = 11;
EXPCds[0].intSDSHfirst_day = 0;
EXPCds[0].intSDSHlast_day = 9;
EXPCds[0].blnSDSHcan_shout = true;
ZIP_Sched_Agent[] dsAgent = EXPCds[0].SDSHagent;
   int intNoAgents = EXPCds[0].intSDSHno_agents;
for ( int intAgent = 0; intAgent < intNoAgents; intAgent++ )
{
// initialise array
dsAgent[intAgent] = new ZIP_Sched_Agent();
dsAgent[intAgent].intAGSHno_units = 1;
}
dsAgent[0].dblAGSHlimit[0] = 3.25; //3.25; // 2.00; // 3.25;
dsAgent[1].dblAGSHlimit[0] = 3.00; //3.00; // 2.00; // 3.00;
dsAgent[2].dblAGSHlimit[0] = 2.75; //2.75; // 2.00; // 2.75;
dsAgent[3].dblAGSHlimit[0] = 2.50; //2.50; // 2.00; // 2.50;
dsAgent[4].dblAGSHlimit[0] = 2.25; //2.25; // 2.00; // 2.25;
dsAgent[5].dblAGSHlimit[0] = 2.00; //2.00;
dsAgent[6].dblAGSHlimit[0] = 1.75; //1.75; // 2.00; // 1.75;
dsAgent[7].dblAGSHlimit[0] = 1.50; //1.50; // 2.00; // 1.50;
dsAgent[8].dblAGSHlimit[0] = 1.25; //1.25; // 2.00; // 1.25;
dsAgent[9].dblAGSHlimit[0] = 1.00; //1.00; // 2.00; // 1.00;
dsAgent[10].dblAGSHlimit[0] = 0.75; //0.75; // 2.00; // 0.75;
// set values for all supply schedule parameters
EXPCss[0].intSDSHno_agents = 11;
EXPCss[0].intSDSHfirst_day = 0;
EXPCss[0].intSDSHlast_day = 9;
```

```
EXPCss[0].blnSDSHcan_shout = true;
ZIP_Sched_Agent[] ssAgent = EXPCss[0].SDSHagent;
intNoAgents = EXPCss[0].intSDSHno_agents;
for ( int intAgent = 0; intAgent < intNoAgents; intAgent++ )
{
// initialise array
ssAgent[intAgent] = new ZIP_Sched_Agent();
ssAgent[intAgent].intAGSHno_units = 1;
}
ssAgent[0].dblAGSHlimit[0] = 0.75; // 0.75; // 0.50; // 2.00;
ssAgent[1].dblAGSHlimit[0] = 1.00; // 1.00; // 0.50; // 2.00;
ssAgent[2].dblAGSHlimit[0] = 1.25; // 1.25; // 0.50; // 2.00;
ssAgent[3].dblAGSHlimit[0] = 1.50; // 1.50; // 0.50; // 2.00;
ssAgent[4].dblAGSHlimit[0] = 1.75; // 1.75; // 0.50; // 2.00;
ssAgent[5].dblAGSHlimit[0] = 2.00; // 2.00; // 0.50; // 2.00;
ssAgent[6].dblAGSHlimit[0] = 2.25;
ssAgent[7].dblAGSHlimit[0] = 2.50;
ssAgent[8].dblAGSHlimit[0] = 2.75;
ssAgent[9].dblAGSHlimit[0] = 3.00;
ssAgent[10].dblAGSHlimit[0] = 3.25;
/*
// set values for all demand schedule parameters
EXPCds[1].intSDSHno_agents = 11;
EXPCds[1].intSDSHfirst_day = 10;
EXPCds[1].intSDSHlast_day = 14;
EXPCds[1].blnSDSHcan_shout = true;
ZIP_Sched_Agent[] dsAgent1 = EXPCds[1].SDSHagent;
   int intNoAgents1 = EXPCds[1].intSDSHno_agents;
for ( int intAgent = 0; intAgent < intNoAgents1; intAgent++ )
{
// initialise array
dsAgent1[intAgent] = new ZIP_Sched_Agent();
dsAgent1[intAgent].intAGSHno_units = 1;
}
dsAgent1[0].dblAGSHlimit[0] = 3.75; //3.25; // 2.00; // 3.25;
dsAgent1[1].dblAGSHlimit[0] = 3.50; //3.00; // 2.00; // 3.00;
dsAgent1[2].dblAGSHlimit[0] = 3.25; //2.75; // 2.00; // 2.75;
dsAgent1[3].dblAGSHlimit[0] = 3.00; //2.50; // 2.00; // 2.50;
dsAgent1[4].dblAGSHlimit[0] = 2.75; //2.25; // 2.00; // 2.25;
dsAgent1[5].dblAGSHlimit[0] = 2.50; //2.00;
dsAgent1[6].dblAGSHlimit[0] = 2.25; //1.75; // 2.00; // 1.75;
dsAgent1[7].dblAGSHlimit[0] = 2.00; //1.50; // 2.00; // 1.50;
dsAgent1[8].dblAGSHlimit[0] = 1.75; //1.25; // 2.00; // 1.25;
dsAgent1[9].dblAGSHlimit[0] = 1.50; //1.00; // 2.00; // 1.00;
dsAgent1[10].dblAGSHlimit[0] = 1.25; //0.75; // 2.00; // 0.75;
*/
}
}
```

C.1.7 ZIP_GA.java

Description: The GA used to evolve the ZIP trading agent parameter sets.

```
// Class: ZIP_GA.java
// ALGORITHM:
// fill population with random values
// for n times round generation loop
//  evaluate fitness of all genomes in the population
//  select preferentially the fitter ones as parents
//  for <intPopSize> times round repro loop
//  pick 2 from parental pool
// recombine to make 1 offspring
// mutate the offspring
//  end repro loop
// throw away parental generation and replace with offspring
// end generation loop
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.text.DecimalFormat;
import java.util.Random;
public class ZIP_GA
{
PrintStream psDAY, psTRA, psGA, psGEN;
// genome consists of eight real-valued parameters
int intGenSize = 8;
int intPopSize = 30;
public static int ZERO = 0;
public static int EASY = 1;
public static int HARD = 2;
public void GA( int intGARun, String strInitConds )
{
Random random = new Random();
//random.setSeed( 500 );
String strFileName1;
String strFileName2;
// ------------------------------------------------------------------
double dblPop[][] = new double[intGenSize][intPopSize];
double dblNewPop[][] = new double[intGenSize][intPopSize];
double dblFitnessScore[] = new double[intPopSize];
double dblFittestGen[] = new double[intGenSize];
double dblCrossoverProb = 0.125;
double dblMutationProb = 0.3;
int intNoOfGens = 200;
int intGen = 1;
// number of ZIP_Sim experiments to carry out for each individual
int intNoExps = 50;
int intInitConds = EASY;
strFileName1 = "ZIP_GA_EASY_ELITE_GEN_exp" + intGARun + ".txt";
strFileName2 = "ZIP_GA_EASY_POP_SCORES_exp" + intGARun + ".txt";
if ( strInitConds.equals( "zero" ) )
{
intInitConds = ZERO;
strFileName1 = "ZIP_GA_ZERO_ELITE_GEN_exp" + intGARun + ".txt";
strFileName2 = "ZIP_GA_ZERO_POP_SCORES_exp" + intGARun + ".txt";
}
else if ( strInitConds.equals( "hard" ) )
{
intInitConds = HARD;
strFileName1 = "ZIP_GA_HARD_ELITE_GEN_exp" + intGARun + ".txt";
strFileName2 = "ZIP_GA_HARD_POP_SCORES_exp" + intGARun + ".txt";
}
try
{
psGA = new PrintStream( new FileOutputStream( strFileName1 ) );
psGEN = new PrintStream( new FileOutputStream( strFileName2 ) );
}
catch( Exception exception )
```

```
{
System.out.println( "IOException: " + exception );
System.exit( 0 );
}
psGA.println( strInitConds + " initial conditions" );
psGA.println(
"LR_MIN LR_MAX MM_MIN MM_MAX PR_MIN PR_MAX TR_R TR_A"
+ " GEN BEST FITNESS" );
try
{
// initialise file to write day data to
psDAY = new PrintStream( new FileOutputStream(
"ZIP_DAY_exp" + intGARun + ".txt" ) );
//+ "_gen" + intGen
// initialise file to write trade data to
psTRA = new PrintStream( new FileOutputStream(
"ZIP_TRA_exp" + intGARun + ".txt" ) );
//"_gen" + intGen +
}
catch( Exception exception )
{
System.out.println( "IOException: " + exception );
System.exit( 0 );
}
// ---------------------------------------------------------------
// FILL POPULATION WITH INITIAL PARAMETER VALUES
// find initial parameter values for hard initial population
// genomes (range [0.00, 0.25])
double dblHardRand1 = random.nextDouble() * 0.25;
// (range [0.75, 1.00])
double dblHardRand2 = ( random.nextDouble() * 0.25 ) + 0.75;
// (the whole population starts with the same 'easy', 'zero' or
// 'hard' values)
for ( int intPopPos = 0; intPopPos < intPopSize; intPopPos++ )
{
for ( int intGenPos = 0; intGenPos < intGenSize; intGenPos++ )
{
if ( intInitConds == ZERO )
{
dblPop[0][intPopPos] = 0.0;
dblPop[1][intPopPos] = 0.0;
dblPop[2][intPopPos] = 0.0;
dblPop[3][intPopPos] = 0.0;
dblPop[4][intPopPos] = 0.0;
dblPop[5][intPopPos] = 0.0;
dblPop[6][intPopPos] = 0.0;
dblPop[7][intPopPos] = 0.0;
}
else if ( intInitConds == EASY )
{
dblPop[0][intPopPos] = 0.10;
dblPop[1][intPopPos] = 0.40;
dblPop[2][intPopPos] = 0.00;
dblPop[3][intPopPos] = 0.10;
dblPop[4][intPopPos] = 0.05;
dblPop[5][intPopPos] = 0.30;
dblPop[6][intPopPos] = 0.05;
dblPop[7][intPopPos] = 0.05;
}
else if ( intInitConds == HARD )
{
dblPop[0][intPopPos] = 0.75;
dblPop[1][intPopPos] = dblHardRand1;
dblPop[2][intPopPos] = 0.75;
dblPop[3][intPopPos] = dblHardRand1;
dblPop[4][intPopPos] = 0.75;
dblPop[5][intPopPos] = dblHardRand1;
dblPop[6][intPopPos] = dblHardRand2;
```

```
dblPop[7][intPopPos] = dblHardRand2;
}
}
}
// --------------------------------------------------------------
// GENERATIONAL LOOP
do
{
// --------------------------------------------------------------
// EVALUATE FITNESS OF GENOMES IN POPULATION
dblFitnessScore = fitness_function( dblPop, dblFitnessScore,
intNoExps, random, intGen, intGARun );
// --------------------------------------------------------------
// (OUTPUT GA GENERATION DATA)
System.out.print( "." );
if ( ( intGen % 10 ) == 0 )
{
System.out.println();
}
// (set to a high value as minimising)
double dblFittestScore = 100000.0;
// identify fittest genome for output purposes (and for elitism)
for ( int intPopPos = 0; intPopPos < intPopSize; intPopPos++ )
{
// (MINIMISING HERE)
if ( dblFitnessScore[intPopPos] < dblFittestScore )
{
dblFittestScore = dblFitnessScore[intPopPos];
for ( int intGenPos = 0; intGenPos < intGenSize;
intGenPos++ )
{
dblFittestGen[intGenPos] = dblPop[intGenPos][intPopPos];
}
}
}
DecimalFormat df = new DecimalFormat( "0.00000" );
// output fittest genome to console
for ( int intGenPos = 0; intGenPos < intGenSize; intGenPos++ )
{
psGA.print( df.format( dblFittestGen[intGenPos] ) + " " );
}
psGA.println( intGen + " " + dblFittestScore );
// output the entire population to a different file
for ( int intPopPos = 0; intPopPos < intPopSize; intPopPos++ )
{
psGEN.print( dblFitnessScore[intPopPos] + " " );
}
psGEN.println();
// --------------------------------------------------------------
// SELECT PREFERENTIALLY FITTER GENOMES AS PARENTS
double[] dblGenome1 = new double[intGenSize];
double[] dblGenome2 = new double[intGenSize];
double[] dblGenome3 = new double[intGenSize];
// --------------------------------------------------------------
// REPRODUCTION LOOP (for intPopSize times)
// create a new population
for ( int intPopPos = 0; intPopPos < intPopSize; intPopPos++ )
{
// randomly select three (unique) genomes from existing
// population
// generate a first random genome selection
int intSelectedGen1 = random.nextInt( intPopSize );
int intSelectedGen2 = intSelectedGen1;
// generate a second random genome selection not the same as
// the first
while ( intSelectedGen2 == intSelectedGen1 )
{
intSelectedGen2 = random.nextInt( intPopSize );
}
```

```
int intSelectedGen3 = intSelectedGen1;
// generate a third random genome selection not the same as the
// first two
while ( intSelectedGen3 == intSelectedGen1 )
{
intSelectedGen3 = intSelectedGen2;
while ( intSelectedGen3 == intSelectedGen2 )
{
intSelectedGen3 = random.nextInt( intPopSize );
}
}
// get the randomly selected genomes from the population
for ( int intGenPos = 0; intGenPos < intGenSize; intGenPos++ )
{
dblGenome1[intGenPos] = dblPop[intGenPos][intSelectedGen1];
dblGenome2[intGenPos] = dblPop[intGenPos][intSelectedGen2];
dblGenome3[intGenPos] = dblPop[intGenPos][intSelectedGen3];
}
// identify the fittest two (become parents) of these three
// (discard most unfit)
// fittest parent genome becomes Vmom, other parent genome Vdad
double dblFitness1 = dblFitnessScore[intSelectedGen1];
double dblFitness2 = dblFitnessScore[intSelectedGen2];
double dblFitness3 = dblFitnessScore[intSelectedGen3];
double dblMom = 100000.0;
double dblDad = 100000.0;
if ( dblFitness1 < dblMom )
{
dblMom = dblFitness1;
if ( dblFitness2 < dblMom )
{
dblMom = dblFitness2;
}
if ( dblFitness3 < dblMom )
{
dblMom = dblFitness3;
}
}
int intMaPos = 0;
int intPaPos = 0;
// if dblFitness1 was the fittest
if ( dblFitness1 == dblMom )
{
intMaPos = intSelectedGen1;
// identify fittest between dblFitness2 and dblFitness3
if ( dblFitness2 < dblFitness3 )
{
dblDad = dblFitness2;
intPaPos = intSelectedGen2;
}
else
{
dblDad = dblFitness3;
intPaPos = intSelectedGen3;
}
}
// if dblFitness2 was the fittest
else if ( dblFitness2 == dblMom )
{
intMaPos = intSelectedGen2;
// identify fittest between dblFitness1 and dblFitness3
if ( dblFitness1 < dblFitness3 )
{
dblDad = dblFitness1;
intPaPos = intSelectedGen1;
}
else
{
dblDad = dblFitness3;
intPaPos = intSelectedGen3;
}
}
// if dblFitness3 was the fittest
else
```

```
{
intMaPos = intSelectedGen3;
// identify fittest between dblFitness1 and dblFitness2
if ( dblFitness1 < dblFitness2 )
{
dblDad = dblFitness1;
intPaPos = intSelectedGen1;
}
else
{
dblDad = dblFitness2;
intPaPos = intSelectedGen2;
}
}
// CROSSOVER: recombine to make 1 offspring
// begin by copying the first element of the Ma genome into
// new Pop
dblNewPop[0][intPopPos] = dblPop[0][intMaPos];
boolean blnParent = true;
// uniform recombination of ma and pa genomes
// (with probability dblCrossoverProb)
for ( int intGenPos = 1; intGenPos < intGenSize;
intGenPos++ )
{
if ( random.nextDouble() < dblCrossoverProb )
{
// change parent
if ( blnParent == true )
{
dblNewPop[intGenPos][intPopPos] =
dblPop[intGenPos][intPaPos];
blnParent = false;
}
else
{
dblNewPop[intGenPos][intPopPos] =
dblPop[intGenPos][intMaPos];
blnParent = true;
}
}
else
{
// stay with same parent
if ( blnParent == true )
{
dblNewPop[intGenPos][intPopPos] =
dblPop[intGenPos][intMaPos];
}
else
{
dblNewPop[intGenPos][intPopPos] =
dblPop[intGenPos][intPaPos];
}
}
}
// MUTATION: mutate the offspring
for ( int intGenPos = 0; intGenPos < intGenSize;
intGenPos++ )
{
// (mutation always carried out)
// generate a random number in range [-0.05,0.05]
double dblRandMut = ( random.nextDouble() * 0.10 ) - 0.05;
dblNewPop[intGenPos][intPopPos] += dblRandMut;
// clip to ensure value in range [0,1]
if ( dblNewPop[intGenPos][intPopPos] > 1.0 )
{
dblNewPop[intGenPos][intPopPos] = 1.0;
}
if ( dblNewPop[intGenPos][intPopPos] < 0.0 )
{
dblNewPop[intGenPos][intPopPos] = 0.0;
```

```
}
}
}
// END REPRODUCTION LOOP
// ------------------------------------------------------------------
// ELITISM: retain best genome of this generation (put into first
// slot of new population)
for ( int intGenPos = 0; intGenPos < intGenSize; intGenPos++ )
{
dblNewPop[intGenPos][0] = dblFittestGen[intGenPos];
}
// throw away parental generation and replace with offspring
for ( int intPopPos = 0; intPopPos < intPopSize; intPopPos++ )
{
for ( int intGenPos = 0; intGenPos < intGenSize; intGenPos++ )
{
dblPop[intGenPos][intPopPos] =
dblNewPop[intGenPos][intPopPos];
}
}
intGen++;
}
// END GENERATION LOOP
while( intGen <= intNoOfGens );
// ------------------------------------------------------------------
}
// ------------------------------------------------------------------
// FITNESS_FUNCTION: returns a fitness score (hamming distance to
// a target string)
public double[] fitness_function( double dblPop[][],
double dblFitnessScore[], int intNoExps, Random random,
int intGen, int intGARun )
{
// call main function
ZIP_Sim sim = new ZIP_Sim();
// iterate through each member of the population
for ( int intPopPos = 0; intPopPos < intPopSize; intPopPos++ )
{
if (( intGen == 1 ) || ( intGen == 200 ))
{
psDAY.println();
psDAY.println( "---------------------------------------" );
psDAY.println( "POPULATION MEMBER: " + intPopPos );
psDAY.println();
psTRA.println();
psTRA.println( "---------------------------------------" );
psTRA.println( "POPULATION MEMBER: " + intPopPos );
psTRA.println();
}
dblFitnessScore[intPopPos] = sim.SIM_main( intNoExps,
dblPop[0][intPopPos], dblPop[1][intPopPos],
dblPop[2][intPopPos], dblPop[3][intPopPos],
dblPop[4][intPopPos], dblPop[5][intPopPos],
dblPop[6][intPopPos], dblPop[7][intPopPos], intGen,
intPopPos, random, psDAY, psTRA );
}
return dblFitnessScore;
}
// ------------------------------------------------------------------
public static void main ( String args[] )
{
ZIP_GA ga = new ZIP_GA();
try
{
String strInitConds = args[0];
if (( strInitConds.equals( "easy" )) ||
( strInitConds.equals( "zero" )) ||
( strInitConds.equals( "hard" )))
{
```

```
int intNoGARuns = 50;
for ( int intGARun = 1; intGARun <= intNoGARuns;
intGARun++ )
{
ga.GA( intGARun, strInitConds );
}
}
else
{
System.out.println(
"usage: java ZIP_GA <init_conds> (init_conds = easy, "
+ "zero or hard)" );
}
}
catch ( ArrayIndexOutOfBoundsException exception )
{
System.out.println(
"usage: java ZIP_GA <init_conds> (where init_conds = "
+ "easy, zero or hard)" );
}
}
// ---------------------------------------------------------------
}
```

C.1.8 ZIP_Sched_Agent.java

Description: Data associated with an agents number of units and buy / sell limits.

```
// Class:  ZIP_Sched_Agent.java
// (used by ZIP_Sched_Agent, ZIP_Exp_Control and ZIP_Exp_Params)
public class ZIP_Sched_Agent implements ZIP_Constants
{
// how many units the agent has / wants
public int intAGSHno_units;
// limit price of each unit
public double[] dblAGSHlimit = new double[MAX_UNITS];
}
```

C.1.9 ZIP_Sched_SD.java

Description: Data associated with a supply and demand schedule.

```
// Class:  ZIP_Sched_SD.java
// (used by ZIP_Exp_Control and ZIP_Exp_Params)
public class ZIP_Sched_SD implements ZIP_Constants
{
// number of agents involved
public int intSDSHno_agents;
// first day this schedule applies to
public int intSDSHfirst_day;
// last day this schedule applies to
public int intSDSHlast_day;
// 0 = silent traders, 1 = can shout
public boolean blnSDSHcan_shout;
// details of individual agents
public ZIP_Sched_Agent[] SDSHagent = new ZIP_Sched_Agent[MAX_AGENTS];
}
```

C.1.10 ZIP_SD_Vis.java

Description: Calculates both the underlying (theoretical) and apparent (actual) equilibrium price, equilibrium quantity and maximum available surplus.

```
// Class:  ZIP_SD_Vis.java
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.text.DecimalFormat;
public class ZIP_SD_Vis implements ZIP_Constants
{
private static int MAX_PRICES = MAX_AGENTS * MAX_UNITS;
private static int MAX_POINTS = MAX_PRICES * 3;
// signifies no equilibrium price / quantity
private static int NULL_EQ = -1;
public double dblEqPrice;
public int intIQuant;
public double dblSurplus;
// ---------------------------------------------------------------
// SDVIS_SUP_DEM: returns either the UNDERLYING (theoretical) or
// APPARENT (actual) values of EQUILIBRIUM PRICE, EQUILIBRIUM QUANTITY,
// and MAXIMUM AVAILABLE SURPLUS. (called by ZIP_Sim.SIM_main,
// ZIP_Sim.SIM_Trade and ZIP_Sim.SIM_init_day)
public void SDVIS_sup_dem( int intNoSellers, ZIP_Agent[] agent_s,
int intNoBuyers, ZIP_Agent[] agent_b, int intMaxTrades,
double TMPdblEqPrice, int TMPintIQuant, double TMPdblSurplus,
int intField, double dblBounds, boolean blnOutput,
PrintStream psSDO )
{
dblEqPrice = TMPdblEqPrice;
intIQuant = TMPintIQuant;
dblSurplus = TMPdblSurplus;
// used to set maximum quantity on graph
int intMaxNo;
// agent indices
int intAgent;
// seller indices
int intSeller;
// buyer indices
int intBuyer;
// unit quantity indices
int intQuantity;
// flag for whether an intersect point exists
boolean blnNoIntersect;
// flag for whether an intersect point has been found
boolean blnNotFound;
// seller limit and quote prices
double[][] dblSellerLimPr = new double[MAX_PRICES][2];
// buyer limit and quote prices
double[][] dblBuyerLimPr = new double[MAX_PRICES][2];
// profit
double dblProfit;
// maximum available surplus
double dblTotalSurplus;
// maximum price
double dblMaxPrice = 0.0;
// minimum price
double dblMinPrice = 0.0;
// declarations for S/D visualisation
// point index
int intVISpoint_index;
// minimum quantity on the graph
int intVISmin_quantity;
// maximum quantity on the graph
int intVISmax_quantity;
// (requires initialisation by java)
int intVISdx = 0;
```

```
// (requires initialisation by java)
int intVISdy = 0;
// (requires initialisation by java)
int intVIStx = 0;
// (requires initialisation by java)
int intVISty = 0;
// (requires initialisation by java)
int intVISmin_y = 0;
int intVISfy;
// (requires initialisation by java)
int intVISmax_y = 0;
// coordinate points in polyline etc.
int intVIScoords[][] = new int[MAX_POINTS][2];
String strVISlabel;
// initially set equilibrium price to -1.0
dblEqPrice = -1.0;
// initially set equilibrium quantity to -1
intIQuant = NULL_EQ;
// if there are too many units for this function
if ((( intNoBuyers * MAX_UNITS ) > MAX_PRICES ) ||
(( intNoSellers * MAX_UNITS ) > MAX_PRICES ))
{
psSDO.println(
"FAIL: Too many units in SDVIS_sup_dem() -- recompile" );
System.out.println(
"FAIL: Too many units in SDVIS_sup_dem() -- recompile" );
System.exit( 0 );
}
// ------------------------------------------------------------------
intSeller = 0;
for ( intAgent = 0; intAgent < intNoSellers; intAgent++ )
{
// if SELLER active
if ( agent_s[intAgent].blnAGNTactive )
{
for ( intQuantity = 0; intQuantity <
agent_s[intAgent].dblAGNTquantity; intQuantity++ )
{
// get the SELLERS limit price
dblSellerLimPr[intSeller][0] =
agent_s[intAgent].dblAGNTlimit;
// get the SELLERS price
dblSellerLimPr[intSeller][1] =
agent_s[intAgent].dblAGNTprice;
// if the first seller agent
if ( intSeller == 0 )
{
// set the minimum and maximum price variables...
dblMaxPrice = agent_s[intAgent].dblAGNTprice;
dblMinPrice = agent_s[intAgent].dblAGNTlimit;
}
else
{
// ...or update the minimum and maximum price variables
if ( agent_s[intAgent].dblAGNTprice > dblMaxPrice )
{
dblMaxPrice = agent_s[intAgent].dblAGNTprice;
}
if ( agent_s[intAgent].dblAGNTlimit < dblMinPrice )
{
dblMinPrice = agent_s[intAgent].dblAGNTlimit;
}
}
// go to the next seller agent
intSeller++;
}
}
}
// ------------------------------------------------------------------
```

```
intBuyer = 0;
for ( intAgent = 0; intAgent < intNoBuyers; intAgent++ )
{
// if buyer active
if ( agent_b[intAgent].blnAGNTactive )
{
for ( intQuantity = 0; intQuantity <
agent_b[intAgent].dblAGNTquantity;
intQuantity++ )
{
// get the BUYERS limit price
dblBuyerLimPr[intBuyer][0] = agent_b[intAgent].dblAGNTlimit;
// get the BUYERS price
dblBuyerLimPr[intBuyer][1] = agent_b[intAgent].dblAGNTprice;
// update the minimum and maximum price variables
if ( agent_b[intAgent].dblAGNTlimit > dblMaxPrice )
{
dblMaxPrice = agent_b[intAgent].dblAGNTlimit;
}
if ( agent_b[intAgent].dblAGNTprice < dblMinPrice )
{
dblMinPrice = agent_b[intAgent].dblAGNTprice;
}
// go to the next buyer agent
intBuyer++;
}
}
}
// sort buyer and seller quote prices (call bubble sort method)
SDVIS_bubble_sort( true, intField, intBuyer, dblBuyerLimPr, psSDO );
SDVIS_bubble_sort( false, intField, intSeller, dblSellerLimPr,
psSDO );
// ---------------------------------------------------------------------
// (VIS) identify the maximum quantity needed on the graph
if ( intSeller > intBuyer )
{
intMaxNo = intSeller;
}
else
{
intMaxNo = intBuyer;
}
// (VIS) set the minimum and maximum quantities on the graph
intVISmin_quantity = 1;
intVISmax_quantity = intMaxNo;
if ( blnOutput )
{
psSDO.println( "SDVIS_sup_dem: Maximum trades: " + intMaxTrades );
psSDO.println( "SDVIS_sup_dem: Minimum price: " + dblMinPrice +
", Maximum price: " + dblMaxPrice + ", Minimum Quantity: " +
intVISmin_quantity + ", Maximum Quantity: "
+ intVISmax_quantity );
psSDO.println();
}
// (VIS) (-1.0 == null)
if ( dblBounds != -1.0 )
{
// autoscaling is off
intVISmin_quantity = (int) dblBounds;
intVISmax_quantity = (int) ( dblBounds + 1 );
dblMinPrice = ( dblBounds + 2 );
dblMaxPrice = ( dblBounds + 3 );
if ( blnOutput )
{
psSDO.println(
"SDVIS_sup_dem: Autoscaling is OFF. Bounds are:" );
psSDO.println( "SDVIS_sup_dem: Minimum Price: " + dblMinPrice
+ ", Maximum Price: " + dblMaxPrice + ", Minimum Quantity: "
+ intVISmin_quantity + ", Maximum Quantity: "
```

```
+ intVISmax_quantity );
psSDO.println();
}
}
// initialise maximum available surplus
dblTotalSurplus = 0.0;
// ------------------------------------------------------------------
if ( dblSellerLimPr[0][intField] > dblBuyerLimPr[0][intField] )
{
// lowest selling price is larger than the highest bidding price
blnNoIntersect = true;
}
else
{
// FIND INTERSECT POINT
blnNoIntersect = false;
blnNotFound = true;
for ( intQuantity = 0; intQuantity < intMaxNo; intQuantity++ )
{
// intersection?
dblProfit = dblBuyerLimPr[intQuantity][intField] -
dblSellerLimPr[intQuantity][intField];
if ( blnNotFound )
{
if ( dblSellerLimPr[intQuantity][intField] >
dblBuyerLimPr[intQuantity][intField] )
{
// straightforward intersect
dblEqPrice = ( dblSellerLimPr[intQuantity - 1][intField]
+ dblBuyerLimPr[intQuantity - 1][intField] ) / 2.0;
intIQuant = intQuantity;
// intersect found
blnNotFound = false;
}
else
{
if (( ( intQuantity + 1 ) == intSeller ) &&
( ( intQuantity + 1 ) == intBuyer ))
{
// last buyer and seller
dblEqPrice = ( dblSellerLimPr[intQuantity][intField]
+ dblBuyerLimPr[intQuantity][intField] ) / 2.0;
intIQuant = ( intQuantity + 1 );
if ( intQuantity < intMaxTrades )
{
dblTotalSurplus += dblProfit;
}
blnNotFound = false;
}
else
{
if (( intQuantity + 1 ) == intSeller )
{
// run out of active sellers but still some
// buyers
dblEqPrice = ( dblBuyerLimPr[intQuantity][intField]
+ dblBuyerLimPr[intQuantity + 1][intField] )
/ 2.0;
intIQuant = ( intQuantity + 1 );
if ( intQuantity < intMaxTrades )
{
dblTotalSurplus += dblProfit;
}
blnNotFound = false;
}
else
{
if (( intQuantity + 1 ) == intBuyer )
{
// run out of active buyers but still some
// sellers
```

```
dblEqPrice = (
dblSellerLimPr[intQuantity][intField]
+ dblSellerLimPr[intQuantity + 1][intField] )
/ 2.0;
intIQuant = ( intQuantity + 1 );
if ( intQuantity < intMaxTrades )
{
dblTotalSurplus += dblProfit;
}
blnNotFound = false;
}
}
}
}
if ( blnNotFound )
{
if ( intQuantity < intMaxTrades )
{
dblTotalSurplus += dblProfit;
}
}
}
if ( blnOutput )
{
//psv.println( "Quantity: " + ( intQuantity + 1 ) );
if ( intQuantity < intSeller )
{
psSDO.println( "SDVIS_sup_dem: Supply: "
+ dblBuyerLimPr[intQuantity][intField] );
}
else
{
psSDO.println( " " );
}
if ( intQuantity < intBuyer )
{
psSDO.println( "SDVIS_sup_dem: Demand: "
+ dblBuyerLimPr[intQuantity][intField] );
}
else
{
psSDO.println( " " );
}
psSDO.println( "SDVIS_sup_dem: Profit: " + dblProfit
+ ", Total Surplus " + dblTotalSurplus );
}
}
dblSurplus = dblTotalSurplus;
if ( blnOutput )
{
psSDO.println();
switch ( intField )
{
case 0: // (EQ_THEORY)
psSDO.print( "SDVIS_sup_dem: THEORETICAL" );
break;
case 1: // (EQ_ACTUAL)
psSDO.print( "SDVIS_sup_dem: ACTUAL" );
break;
default:
psSDO.print( "FAIL: bad field " + intField +
" in supdem" );
System.out.println( "FAIL: bad field " + intField
+ " in supdem" );
System.exit( 0 );
}
psSDO.println( " EQUILIBRIUM PRICE: " + dblEqPrice
+ " at EQUILIBRIUM QUANTITY: " + intIQuant +
", MAXIMUM SURPLUS: " + dblSurplus );
psSDO.println();
```

```
}
}
}
// ---------------------------------------------------------------------
// bubble sort (called by ZIP_SD_Vis.SDVIS_sup_dem)
public void SDVIS_bubble_sort( boolean blnOrder, int intField, int n,
double[][] dblArray, PrintStream psSDO )
{
int i;
int j;
boolean blnSwap;
double t;
if (( intField < 0 ) || ( intField > 1 ))
{
psSDO.println( "FAIL: bad field " + intField + " in sort" );
System.out.println( "FAIL: bad field " + intField
+ " in sort" );
System.exit( 0 );
}
for ( i = 0; i < n; i++ )
{
for ( j = 0; j < i; j++ )
{
if ( blnOrder )
{
if ( dblArray[i][intField] > dblArray[j][intField] )
{
blnSwap = true;
}
else
{
blnSwap = false;
}
}
else
{
if ( dblArray[i][intField] < dblArray[j][intField] )
{
blnSwap = true;
}
else
{
blnSwap = false;
}
}
if ( blnSwap )
{
t = dblArray[i][0];
dblArray[i][0] = dblArray[j][0];
dblArray[j][0] = t;
t = dblArray[i][1];
dblArray[i][1] = dblArray[j][1];
dblArray[j][1] = t;
}
}
}
}
// ---------------------------------------------------------------------
}
```

C.1.11 ZIP_Sim.java

Description: Main simulation class. Initialises ZIP agents and co-ordinates all trading activity (Nb. extra additions for the GA are commented 'GA').

```
// Class:  ZIP_Sim.java
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.text.DecimalFormat;
import java.util.Random;
public class ZIP_Sim implements ZIP_Constants
{
   PrintStream psOUT, psSDO;
   // GLOBAL PARAMETERS
   private int intStatus = 0;
   private double dblPrice_0;
   private double dblSurplus;
   private double dblMaxSurplus;
   private ZIP_Data_Day[] DayData;
   private ZIP_Exp_Control ec;
   // ---------------------------------------------------------------------
   public static void main( String args[] )
   {
    ZIP_Sim sim = new ZIP_Sim();
   try
   {
      int intNoExps = Integer.parseInt( args[0] );
      sim.SIM_main( intNoExps );
   }
   catch ( Exception exception )
   {
     System.out.println( "usage: java ZIP_Sim <no_experiments>" );
   }
   }
   // ---------------------------------------------------------------------
// SIM_MAIN (called by ZIP_Sim.main)
public void SIM_main( int TMPintNoExps )
{
// SIM_MAIN (called by ZIP_Sim.main) (GA -- CONSTRUCTOR)
public double SIM_main( int TMPintNoExps, double dblAGNT_lr_L,
double dblAGNT_lr_H, double dblAGNT_mm_L, double dblAGNT_mm_H,
double dblAGNT_pf_L, double dblAGNT_pf_H, double dblAGNT_tr_R,
double dblAGNT_tr_A, int intGen, int intPopPos, Random random,
PrintStream psDAY, PrintStream psTRA )
{
// initialise output files and output streams
try
{
psOUT = new PrintStream( new FileOutputStream(
"ZIP_OUT.txt" ) );
psSDO = new PrintStream( new FileOutputStream(
"ZIP_SDO.txt" ) );
}
catch( Exception exception )
{
System.out.println( "IOException: " + exception );
}
// ZIP_Sim PARAMETERS
// number of transactions on a day
int intNoTrans;
// day indices
int intDayNo;
// random seed
int intRandSeed;
// number of buyers
int intNoBuyers;
// number of sellers
int intNoSellers;
```

```
// number of trades done in a day
int intNoTrades;
// number of experiments to run
int intNoExps = TMPintNoExps;
// maximum number of trades in a session
int intMaxTrades = 0;
// equilibrium quantity
int intEqQuantity;
// (used in calling ZIP_SD_Vis.SDVIS_sup_dem method)
int intDummy = 0;
// experiment number (indices)
int intExpNo;
// transaction number within a day
int intDayTransNo;
// number of Alpha over Transaction Sequences (in array dblATS)
int[] intNoATS = new int[MAX_TRADES];
// (initialised for use below)
double dblPrice = 0.0;
// sum the s.d. of the transaction (deal) price around the
// equilibrium price
double dblSigmaSum;
// smith's alpha (initialised to send to data_day_update
// method below)
double dblAlpha = 0.0;
// equilibrium price
double dblEqPrice;
// the last deal price
double dblLastPrice = 0.0;
// sum of the price difference between the last deal price
// and the current deal price
double dblPriceDiffSum;
// used in calling ZIP_SD_Vis.SDVIS_sup_dem method
double dblDummy1 = 0.0;
// used in calling ZIP_SD_Vis.SDVIS_sup_dem method
double dblDummy2 = 0.0;
// sum of the deal prices
double dblPriceSum;
// difference between actual gain and theoretical gain
double dblDiff;
// square difference between actual gain and theoretical gain
double dblPD;
// profit dispersal
double dblProfitDisp;
// s.d. of the transaction price around the equilibrium price
// (used to calculate smith's alpha)
double dblPDS;
// alpha over transaction sequence
double dblAlphaTrans;
// alpha over transaction sequence
double[] dblATS = new double[MAX_TRADES];
// can be used to inhibit autoscaling on supdem
double[] dblBoundData = new double[4];
double dblBounds;
// (initialised to send to data_day_update method below)
double dblEfficiency = 0.0;
boolean blnOutput = false;
DayData = new ZIP_Data_Day[MAX_N_DAYS];
ZIP_Data_Trade[][] TradeData =
new ZIP_Data_Trade[MAX_N_DAYS][MAX_TRADES];
// for summarising ats[] over (several?) experiments
ZIP_Day_Stat[] ats_e = new ZIP_Day_Stat[MAX_TRADES];
ZIP_Agent[] buyers = new ZIP_Agent[MAX_AGENTS];
ZIP_Agent[] sellers = new ZIP_Agent[MAX_AGENTS];
double[] TMPdblSum = new double[MAX_N_DAYS];
double[] TMPdblSumSq = new double[MAX_N_DAYS];
int[] TMPintN = new int[MAX_N_DAYS];
ec = new ZIP_Exp_Control();
ZIP_SD_Vis sdvis = new ZIP_SD_Vis();
```

```
// (GA)
double dblExpFitness = 0.0;
// ---------------------------------------------------------------
// ZIP_Sim CODE:
if ( blnOutput )
{
psOUT.println( "ZIP_SIM: " + intNoExps + " experiments" );
}
// GET EXPERIMENT PARAMETERS AND READ S/D SCHEDULES (control data)
ec.EXPC_in( true, psOUT );
// arrays used to store cumulative values of day parameters in
// order to obtain average values over all experiments carried out
double[] dblAlpha_Mean = new double[ec.intEXPCno_days];
double[] dblAlpha_m1SD = new double[ec.intEXPCno_days];
double[] dblAlpha_p1SD = new double[ec.intEXPCno_days];
double[] dblQuant_Mean = new double[ec.intEXPCno_days];
double[] dblQuant_m1SD = new double[ec.intEXPCno_days];
double[] dblQuant_p1SD = new double[ec.intEXPCno_days];
double[] dblEffic_Mean = new double[ec.intEXPCno_days];
double[] dblEffic_m1SD = new double[ec.intEXPCno_days];
double[] dblEffic_p1SD = new double[ec.intEXPCno_days];
double[] dblPrice_Mean = new double[ec.intEXPCno_days];
double[] dblPrice_m1SD = new double[ec.intEXPCno_days];
double[] dblPrice_p1SD = new double[ec.intEXPCno_days];
double[] dblPDisp_Mean = new double[ec.intEXPCno_days];
double[] dblPDisp_m1SD = new double[ec.intEXPCno_days];
double[] dblPDisp_p1SD = new double[ec.intEXPCno_days];
double[] dblVolty_Mean = new double[ec.intEXPCno_days];
double[] dblVolty_m1SD = new double[ec.intEXPCno_days];
double[] dblVolty_p1SD = new double[ec.intEXPCno_days];
// INITIALISE DAY DATA RECORDS (for all days)
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days; intDayNo++ )
{
// (initialise array)
DayData[intDayNo] = new ZIP_Data_Day();
DayData[intDayNo].data_day_init();
}
// (initialise TradeData array)
for ( intDayNo = 0; intDayNo < MAX_N_DAYS; intDayNo++ )
{
for ( intDayTransNo = 0; intDayTransNo < MAX_TRADES;
intDayTransNo++ )
{
TradeData[intDayNo][intDayTransNo] = new ZIP_Data_Trade();
}
}
// initialise alpha parameter related arrays (for visualisation)
for ( intDayTransNo = 0; intDayTransNo < MAX_TRADES;
intDayTransNo++ )
{
intNoATS[intDayTransNo] = 0;
dblATS[intDayTransNo] = 0.0;
ats_e[intDayTransNo] = new ZIP_Day_Stat();
ats_e[intDayTransNo].intN = 0;
ats_e[intDayTransNo].dblSum = 0.0;
ats_e[intDayTransNo].dblSumSq = 0.0;
}
psTRA.println( "ZIP_SIM TRADE DATA" );
psDAY.println( "ZIP_SIM DAY DATA" );
psSDO.println( "ZIP_SIM SUPPLY AND DEMAND DATA" );
// ---------------------------------------------------------------
// EXPERIMENT LOOP
for ( intExpNo = 0; intExpNo < intNoExps; intExpNo++ )
{
for ( int intBuyer = 0; intBuyer < MAX_AGENTS; intBuyer++ )
{
// INITIALISE BUYER AGENTS (parameters and profit value)
```

```
buyers[intBuyer] = new ZIP_Agent();
// (GA) SET BUYER AGENT GA PARAMETERS
buyers[intBuyer].dblAGNT_lr_L = dblAGNT_lr_L;
buyers[intBuyer].dblAGNT_lr_H = dblAGNT_lr_H;
buyers[intBuyer].dblAGNT_mm_L = dblAGNT_mm_L;
buyers[intBuyer].dblAGNT_mm_H = dblAGNT_mm_H;
buyers[intBuyer].dblAGNT_pf_L = dblAGNT_pf_L;
buyers[intBuyer].dblAGNT_pf_H = dblAGNT_pf_H;
buyers[intBuyer].dblAGNT_tr_R = dblAGNT_tr_R;
buyers[intBuyer].dblAGNT_tr_A = dblAGNT_tr_A;
buyers[intBuyer].AGNT_init_buyer( intBuyer, blnOutput,
psOUT, random );
}
if ( blnOutput )
{
psOUT.println();
}
for ( int intSeller = 0; intSeller < MAX_AGENTS; intSeller++ )
{
// INITIALISE SELLER AGENTS (parameters and profit value)
sellers[intSeller] = new ZIP_Agent();
// (GA) SET SELLER AGENT GA PARAMETERS
sellers[intSeller].dblAGNT_lr_L = dblAGNT_lr_L;
sellers[intSeller].dblAGNT_lr_H = dblAGNT_lr_H;
sellers[intSeller].dblAGNT_mm_L = dblAGNT_mm_L;
sellers[intSeller].dblAGNT_mm_H = dblAGNT_mm_H;
sellers[intSeller].dblAGNT_pf_L = dblAGNT_pf_L;
sellers[intSeller].dblAGNT_pf_H = dblAGNT_pf_H;
sellers[intSeller].dblAGNT_tr_R = dblAGNT_tr_R;
sellers[intSeller].dblAGNT_tr_A = dblAGNT_tr_A;
sellers[intSeller].AGNT_init_seller( intSeller, blnOutput,
psOUT, random );
}
// ---------------------------------------------------------------
// DAY LOOP
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days; intDayNo++ )
{
// one day or 'trading period'
// set maximum number of trades in this day
intMaxTrades = ec.intEXPCmax_trades;
// (output day and maximum number of trades)
if ( blnOutput )
{
psOUT.println();
psOUT.println( "ZIP_SIM: Day " + intDayNo
+ ": Maximum number of trades: " + MAX_TRADES );
psOUT.println();
}
// INITIALISE ALL DATA STRUCTURES FOR START OF DAY
SIM_init_day( intDayNo, ec, sellers, buyers, blnOutput,
psOUT, sdvis );
// get number of buyer and seller agents
intNoBuyers = ec.EXPCds[ec.intEXPCds].intSDSHno_agents;
intNoSellers = ec.EXPCss[ec.intEXPCss].intSDSHno_agents;
// initialise variables used in calculating trade statistics
// (below)
dblSurplus = 0.0;
intNoTrades = 0;
dblSigmaSum = 0.0;
dblPriceSum = 0.0;
dblPriceDiffSum = 0.0;
// (VIS) (-1.0 counts as null)
dblBounds = -1.0;
dblBoundData[0] = 0;
dblBoundData[1] = 12;
dblBoundData[2] = 0.0;
dblBoundData[3] = 3.75;
// if the first experiment
```

```
if ( intExpNo == 0 )
{
// write a figure of the actual supply and demand curves
sdvis.SDVIS_sup_dem( intNoSellers, sellers, intNoBuyers,
buyers, intMaxTrades, dblDummy1, intDummy, dblDummy2,
EQ_ACTUAL, dblBounds, blnOutput, psSDO );
dblDummy1 = sdvis.dblEqPrice;
intDummy = sdvis.intIQuant;
dblDummy2 = sdvis.dblSurplus;
}
// -------------------------------------------------------------
// TRADING SESSION LOOP
for ( intDayTransNo = 0; intDayTransNo < intMaxTrades;
intDayTransNo++ )
{
// one trading session: either a trade occurs or a fail is
// recorded
if ( blnOutput )
{
psOUT.println();
psOUT.println( "----------------------------------" );
psOUT.println( "ZIP_SIM: DAY: " + intDayNo + " TRADE: "
+ ( intDayTransNo + 1 ) );
psOUT.println( "----------------------------------" );
psOUT.println();
}
// SEE IF A BUYER OR A SELLER CAN BE FOUND WHO WILL ENTER
// INTO A TRADE
SIM_trade( TradeData[intDayNo][intDayTransNo], sellers,
buyers, ec, blnOutput, psOUT, sdvis, random );
// if first experiment print a figure of the actual
// supply and demand curves
if (( blnOutput ) && ( intExpNo == 0 ))
{
sdvis.SDVIS_sup_dem( intNoSellers, sellers, intNoBuyers,
buyers, intMaxTrades, dblDummy1, intDummy, dblDummy2,
EQ_ACTUAL, dblBounds, blnOutput, psSDO );
dblDummy1 = sdvis.dblEqPrice;
intDummy = sdvis.intIQuant;
dblDummy2 = sdvis.dblSurplus;
}
// if a buyer and a seller entered into a trade (a deal
// occured)
// CALCULATE DAY AND TRADE STATISTICS
if ( intStatus == DEAL )
{
// if not the first day transaction
if ( intDayTransNo > 0 )
{
// literally make the last deal price the last deal
// price
dblLastPrice = dblPrice;
}
// now obtain the the current (last) deal price
dblPrice =
TradeData[intDayNo][intDayTransNo].dblTDATdeal_price;
if ( intDayTransNo > 0 )
{
// sum of the price difference between the last deal
// price and the current deal price
dblPriceDiffSum += (( dblPrice - dblLastPrice ) *
( dblPrice - dblLastPrice ));
}
// calculate s.d. of the transaction price around the
// equilibrium price
dblPDS = (( dblPrice - dblPrice_0 ) *
( dblPrice - dblPrice_0 ));
// add alpha over transaction sequence array
dblATS[intNoTrades] += dblPDS;
```

```
// increment number of alpha over transaction sequences
// array
intNoATS[intNoTrades]++;
// increment number of trades count
intNoTrades++;
// sum current deal prices
dblPriceSum += dblPrice;
// sum current s.d. of the transaction (deal) price
// around the equilibrium price
dblSigmaSum += dblPDS;
// DAY STAT: use this to calculate the SMITH'S ALPHA
// parameter
dblAlpha = ( 100 * Math.sqrt( dblSigmaSum
/ intNoTrades )) / dblPrice_0;
// DAY STAT: calculate the EFFICIENCY parameter
dblEfficiency = ( dblSurplus / dblMaxSurplus ) * 100;
if ( blnOutput )
{
psOUT.println( "ZIP_SIM: Day " + intDayNo + " DEAL "
+ intNoTrades + " alpha " + dblAlpha
+ " efficiency " + dblEfficiency );
}
} // end of calculate day/trade statistics
else
{
// give up
if ( intStatus == END_DAY )
{
intDayTransNo = intMaxTrades;
}
}
} // END OF TRADING SESSION (LOOP)
// ------------------------------------------------------------
// UPDATE DATA FOR THIS DAY
// DAY STAT: calculate profit dispersal
dblPD = 0.0;
for ( int intBuyer = 0; intBuyer < intNoBuyers; intBuyer++ )
{
// calculate difference between actual gain and theoretical
// gain
dblDiff = (( buyers[intBuyer].dblAGNTactualgain ) -
( buyers[intBuyer].dblAGNTtheorgain ));
// square this difference
dblPD += ( dblDiff * dblDiff );
}
for ( int intSeller = 0; intSeller < intNoSellers;
intSeller++ )
{
// calculate difference between actual gain and
// theoretical gain
dblDiff = (( sellers[intSeller].dblAGNTactualgain ) -
( sellers[intSeller].dblAGNTtheorgain ));
// square this difference
dblPD += ( dblDiff * dblDiff );
}
// DAY STAT: finish calculating profit dispersal stat
dblProfitDisp = Math.sqrt( ( 1 / ( (double)( intNoBuyers
+ intNoSellers ) ) ) * dblPD );
if ( blnOutput )
{
psOUT.println( "ZIP_SIM: Profit Dispersal = "
+ dblProfitDisp );
psOUT.println();
psOUT.println();
}
// UPDATE DAY DATA
DayData[intDayNo].data_day_update( intNoTrades,
dblPriceSum, dblAlpha, dblProfitDisp, dblEfficiency,
dblPriceDiffSum );
```

```
} // END OF DAY LOOP
// ------------------------------------------------------------
// (VIS) OUTPUT TRADE STATISTICS
// if the first experiment
if ( intExpNo == 0 )
{
DecimalFormat df = new DecimalFormat( "0.0" );
DecimalFormat DF = new DecimalFormat( "0.00" );
double dblDGX = ( 1.0 / ( (double) intMaxTrades ) );
// (VIS) TRADE DATA OUTPUT: PRICE
psTRA.println( " PRICE (n=0 only)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
double dblGX = ( intDayNo + 1 );
double dblSum = DayData[intDayNo].dblSTAT_quant.dblSum;
for ( intDayTransNo = 0; intDayTransNo < dblSum;
intDayTransNo++ )
{
if (
TradeData[intDayNo][intDayTransNo].dblTDATdeal_price
>= 0.0 )
{
psTRA.println( df.format( dblGX )  + " "
+ TradeData[intDayNo][intDayTransNo].dblTDATdeal_price );
}
dblGX += dblDGX;
}
}
// (VIS) TRADE DATA OUTPUT: ACTUAL EQUILIBRIUM PRICE
psTRA.println(
" ACTUAL EQUILIBRIUM PRICE (n=0 output only)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
double dblGX = ( intDayNo + 1 );
double dblSum =
DayData[intDayNo].dblSTAT_quant.dblSum;
for ( intDayTransNo = 0; intDayTransNo < dblSum;
intDayTransNo++ )
{
if (
TradeData[intDayNo][intDayTransNo].intTDATactual_eq_quant
!= NULL_EQ )
{
psTRA.println( df.format( dblGX ) + " " +
DF.format( TradeData[intDayNo][intDayTransNo].dblTDATactual_eq_price ) );
dblGX += dblDGX;
}
}
}
// (VIS) TRADE DATA OUTPUT: THEORETICAL EQUILIBRIUM PRICE
psTRA.println(
" THEORETICAL EQUILIBRIUM PRICE (n=0 output only)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
double dblGX = ( intDayNo + 1 );
double dblSum = DayData[intDayNo].dblSTAT_quant.dblSum;
for ( intDayTransNo = 0; intDayTransNo < dblSum;
intDayTransNo++ )
{
if ( TradeData[intDayNo][intDayTransNo].intTDATtheor_eq_quant
!= NULL_EQ )
{
psTRA.println( df.format( dblGX ) + " " +
TradeData[intDayNo][intDayTransNo].dblTDATtheor_eq_price );
dblGX += dblDGX;
}
```

```
}
}
// (VIS) output this experiments' per-transaction rms
// deviation of deal price from equilibrium
// (not output as same as MEAN below)
}
for ( intDayTransNo = 0; intDayTransNo < intMaxTrades;
intDayTransNo++ )
{
if ( intNoATS[intDayTransNo] > 0 )
{
dblAlphaTrans = Math.sqrt( dblATS[intDayTransNo]
/ intNoATS[intDayTransNo] );
ats_e[intDayTransNo].dblSum += dblAlphaTrans;
ats_e[intDayTransNo].dblSumSq += ( dblAlphaTrans
* dblAlphaTrans );
ats_e[intDayTransNo].intN++;
}
}
if ( blnOutput )
{
psOUT.println( "ZIP_SIM: Experiment " + intExpNo +
" done" );
}
// ---------------------------------------------------------
// (VIS) DAY DATA OUTPUT: plot the mean (and the s.d. for the
// daily stats if more than one experiment) -- for output via
// excel
ZIP_Data_Day ddVIS = new ZIP_Data_Day();
// PRICE
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
TMPdblSum[intDayNo] =
DayData[intDayNo].dblSTAT_price.dblSum;
TMPdblSumSq[intDayNo] =
DayData[intDayNo].dblSTAT_price.dblSumSq;
TMPintN[intDayNo] =
DayData[intDayNo].dblSTAT_price.intN;
}
ddVIS.data_day_stats( "PRICE", TMPdblSum, TMPdblSumSq,
TMPintN, ec.intEXPCno_days, intExpNo, dblPrice_Mean,
dblPrice_m1SD, dblPrice_p1SD, psDAY );
// SMITH'S ALPHA
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
TMPdblSum[intDayNo] =
DayData[intDayNo].dblSTAT_alpha.dblSum;
TMPdblSumSq[intDayNo] =
DayData[intDayNo].dblSTAT_alpha.dblSumSq;
TMPintN[intDayNo] =
DayData[intDayNo].dblSTAT_alpha.intN;
}
ddVIS.data_day_stats( "ALPHA", TMPdblSum, TMPdblSumSq,
TMPintN, ec.intEXPCno_days, intExpNo, dblAlpha_Mean,
dblAlpha_m1SD, dblAlpha_p1SD, psDAY );
// ---------------------------------------------------------
// (GA) CALCULATE FITNESS USING WEIGHTED ALPHA
// PARAMETER OVER 6 DAYS
dblExpFitness += ddVIS.data_day_alpha_fitness( TMPdblSum,
TMPintN );
// ---------------------------------------------------------
// EFFICIENCY
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
TMPdblSum[intDayNo] =
```

```
DayData[intDayNo].dblSTAT_effic.dblSum;
TMPdblSumSq[intDayNo] =
DayData[intDayNo].dblSTAT_effic.dblSumSq;
TMPintN[intDayNo] =
DayData[intDayNo].dblSTAT_effic.intN;
}
ddVIS.data_day_stats( "EFFICIENCY", TMPdblSum, TMPdblSumSq,
TMPintN, ec.intEXPCno_days, intExpNo, dblEffic_Mean,
dblEffic_m1SD, dblEffic_p1SD, psDAY );
// QUANTITY
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
TMPdblSum[intDayNo] =
DayData[intDayNo].dblSTAT_quant.dblSum;
TMPdblSumSq[intDayNo] =
DayData[intDayNo].dblSTAT_quant.dblSumSq;
TMPintN[intDayNo] =
DayData[intDayNo].dblSTAT_quant.intN;
}
ddVIS.data_day_stats( "QUANTITY", TMPdblSum, TMPdblSumSq,
TMPintN, ec.intEXPCno_days, intExpNo, dblQuant_Mean,
dblQuant_m1SD, dblQuant_p1SD, psDAY );
// PROFIT DISPERSAL
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
TMPdblSum[intDayNo] =
DayData[intDayNo].dblSTAT_pdisp.dblSum;
TMPdblSumSq[intDayNo] =
DayData[intDayNo].dblSTAT_pdisp.dblSumSq;
TMPintN[intDayNo] =
DayData[intDayNo].dblSTAT_pdisp.intN;
}
ddVIS.data_day_stats( "PROFIT DISPERSAL", TMPdblSum,
TMPdblSumSq, TMPintN, ec.intEXPCno_days, intExpNo,
dblPDisp_Mean, dblPDisp_m1SD, dblPDisp_p1SD, psDAY );
// VOLATILITY
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
TMPdblSum[intDayNo] =
DayData[intDayNo].dblSTAT_volty.dblSum;
TMPdblSumSq[intDayNo] =
DayData[intDayNo].dblSTAT_volty.dblSumSq;
TMPintN[intDayNo] =
DayData[intDayNo].dblSTAT_volty.intN;
}
ddVIS.data_day_stats( "VOLATILITY", TMPdblSum, TMPdblSumSq,
TMPintN, ec.intEXPCno_days, intExpNo, dblVolty_Mean,
dblVolty_m1SD, dblVolty_p1SD, psDAY );
} // END OF EXPERIMENT LOOP
// ------------------------------------------------------
// plot per-transaction rms deviation of deal price from
// equilibrium, over exp
// mean
psTRA.println( " MEAN" );
for ( intDayTransNo = 0; intDayTransNo < intMaxTrades;
intDayTransNo++ )
{
if ( ats_e[intDayTransNo].intN > 0 )
{
double dblMean = ( ats_e[intDayTransNo].dblSum /
ats_e[intDayTransNo].intN );
psTRA.println( ( intDayTransNo + 1 ) + " " + dblMean );
}
}
```

```
// + 1 standard deviation
psTRA.println( " MEAN + 1 S.D." );
for ( intDayTransNo = 0; intDayTransNo < intMaxTrades;
intDayTransNo++ )
{
if ( ats_e[intDayTransNo].intN > 0 )
{
dblAlphaTrans = ats_e[intDayTransNo].dblSum /
ats_e[intDayTransNo].intN;
psTRA.println( ( intDayTransNo + 1 ) + " " +
( dblAlphaTrans + Math.sqrt((
ats_e[intDayTransNo].dblSumSq /
ats_e[intDayTransNo].intN )
- ( dblAlphaTrans * dblAlphaTrans ))));
}
}
// - 1 standard deviation
psTRA.println( " MEAN - 1 S.D." );
for ( intDayTransNo = 0; intDayTransNo < intMaxTrades;
intDayTransNo++ )
{
if ( ats_e[intDayTransNo].intN > 0 )
{
dblAlphaTrans = ats_e[intDayTransNo].dblSum /
ats_e[intDayTransNo].intN;
psTRA.println( ( intDayTransNo + 1 ) + " " +
( dblAlphaTrans - Math.sqrt((
ats_e[intDayTransNo].dblSumSq /
ats_e[intDayTransNo].intN )
- ( dblAlphaTrans * dblAlphaTrans ))));
}
}
// experiment n as a proportion of n experiments (will be
// 1.0 if 1 experiment)
psTRA.println( " N/NEXPS" );
for ( intDayTransNo = 0; intDayTransNo < intMaxTrades;
intDayTransNo++ )
{
if ( ats_e[intDayTransNo].intN > 0 )
{
double dblNExps = (double) ( ats_e[intDayTransNo].intN
/ intNoExps );
psTRA.println( ( intDayTransNo + 1 ) + " "
+ dblNExps );
}
}
psDAY.println();
psDAY.println( "END OF EXPERIMENTS: DAY DATA AVERAGES" );
psDAY.println( " PRICE MEAN" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblPrice_Mean[intDayNo] / intNoExps ) );
}
psDAY.println( " PRICE MEAN (-1 S.D.)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblPrice_m1SD[intDayNo] / ( intNoExps - 1 ) ) );
}
psDAY.println( " PRICE MEAN (+1 S.D.)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
```

```
+ ( dblPrice_p1SD[intDayNo] / ( intNoExps - 1 ) ) );
}
psDAY.println( " ALPHA MEAN" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblAlpha_Mean[intDayNo] / intNoExps ) );
}
psDAY.println( " ALPHA MEAN (-1 S.D.)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblAlpha_m1SD[intDayNo] / ( intNoExps - 1 ) ) );
}
psDAY.println( " ALPHA MEAN (+1 S.D.)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblAlpha_p1SD[intDayNo] / ( intNoExps - 1 ) ) );
}
psDAY.println( " EFFICIENCY MEAN" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblEffic_Mean[intDayNo] / intNoExps ) );
}
psDAY.println( " EFFICIENCY MEAN (-1 S.D.)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblEffic_m1SD[intDayNo] / ( intNoExps - 1 ) ) );
}
psDAY.println( " EFFICIENCY MEAN (+1 S.D.)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblEffic_p1SD[intDayNo] / ( intNoExps - 1 ) ) );
}
psDAY.println( " QUANTITY MEAN" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblQuant_Mean[intDayNo] / intNoExps ) );
}
psDAY.println( " QUANTITY MEAN (-1 S.D.)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblQuant_m1SD[intDayNo] / ( intNoExps - 1 ) ) );
}
psDAY.println( " QUANTITY MEAN (+1 S.D.)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblQuant_p1SD[intDayNo] / ( intNoExps - 1 ) ) );
}
psDAY.println( " PROFIT DISPERSAL MEAN" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
```

```
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblPDisp_Mean[intDayNo] / intNoExps ) );
}
psDAY.println( " PROFIT DISPERSAL MEAN (-1 S.D.)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblPDisp_m1SD[intDayNo] / ( intNoExps - 1 ) ) );
}
psDAY.println( " PROFIT DISPERSAL MEAN (+1 S.D.)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblPDisp_p1SD[intDayNo] / ( intNoExps - 1 ) ) );
}
psDAY.println( " VOLATILITY MEAN" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblVolty_Mean[intDayNo] / intNoExps ) );
}
psDAY.println( " VOLATILITY MEAN (-1 S.D.)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblVolty_m1SD[intDayNo] / ( intNoExps - 1 ) ) );
}
psDAY.println( " VOLATILITY MEAN (+1 S.D.)" );
for ( intDayNo = 0; intDayNo < ec.intEXPCno_days;
intDayNo++ )
{
psDAY.println( ( intDayNo + 1 ) + " "
+ ( dblVolty_p1SD[intDayNo] / ( intNoExps - 1 ) ) );
}
// (GA)
dblExpFitness = dblExpFitness / intNoExps;
// (GA)
return dblExpFitness;
}
// ------------------------------------------------------------------
// SIM_INIT_DAY: initialise all data structures for start of day
// (called by ZIP_Sim.SIM_main)
public void SIM_init_day( int intDayNo, ZIP_Exp_Control ec,
ZIP_Agent[] sellers, ZIP_Agent[] buyers, boolean blnOutput,
PrintStream psOUT, ZIP_SD_Vis sdvis )
{
int intQ_0 = 0;
double dblEqProfit;
// ------------------------------------------------------------------
// INITIALISE BUYERS
// if first day
if ( intDayNo == 0 )
{
// read the first demand schedule
ec.intEXPCds = 0;
}
// if last day
else if (( intDayNo - 1 ) == (
ec.EXPCds[ec.intEXPCds].intSDSHlast_day ))
{
// previous day was last day on that demand schedule: update
ec.intEXPCds++;
if ( ec.intEXPCds == ec.intEXPCno_ds )
```

```
{
psOUT.println( "FAIL: ran out of demand schedules on day "
+ intDayNo );
System.out.println(
"FAIL: ran out of demand schedules on day " + intDayNo );
System.exit( 0 );
}
}
// set index of currently active demand schedule
int intDemSched = ec.intEXPCds;
// get number of buyer agents from experiment control data (demand
// schedule no. of agents)
int intNoBuyers = ec.EXPCds[intDemSched].intSDSHno_agents;
// MARK ALL BUYERS ACTIVE, SET QUANTITIES AND LIMIT PRICES
for ( int intBuyer = 0; intBuyer < intNoBuyers; intBuyer++ )
{
buyers[intBuyer].dblAGNTquantity =
ec.EXPCds[intDemSched].SDSHagent[intBuyer].intAGSHno_units;
buyers[intBuyer].blnAGNTactive = true;
buyers[intBuyer].dblAGNTactualgain = 0.0;
// (note: only allows for one limit price)
buyers[intBuyer].dblAGNTlimit =
ec.EXPCds[intDemSched].SDSHagent[intBuyer].dblAGSHlimit[0];
// SET THE PRICE OF THE AGENT FROM ITS LIMIT AND PROFIT VALUES
buyers[intBuyer].AGNT_calcPrice();
if ( blnOutput )
{
psOUT.println( "ZIP_SIM.init_day: Buyer " + intBuyer
+ " price " + buyers[intBuyer].dblAGNTprice );
}
}
if ( blnOutput )
{
psOUT.println();
}
// ---------------------------------------------------------------
// INITIALISE SELLERS
// if first day
if ( intDayNo == 0 )
{
// read the first demand schedule
ec.intEXPCss = 0;
}
// if last day
else if (( intDayNo - 1 ) == (
ec.EXPCss[ec.intEXPCss].intSDSHlast_day ))
{
// previous day was last day on that demand schedule: update
ec.intEXPCss++;
if ( ec.intEXPCss == ec.intEXPCno_ss )
{
psOUT.println( "FAIL: ran out of supply schedules on day "
+ intDayNo );
System.out.println(
"FAIL: ran out of supply schedules on day " + intDayNo );
System.exit( 0 );
}
}
// set index of currently active supply schedule
int intSupSched = ec.intEXPCss;
// get number of seller agents from experiment control data (supply
// schedule no. of agents)
int intNoSellers = ec.EXPCss[intSupSched].intSDSHno_agents;
// MARK ALL SELLERS ACTIVE, SET QUANTITIES AND LIMIT PRICES
for ( int intSeller = 0; intSeller < intNoSellers; intSeller++ )
{
sellers[intSeller].dblAGNTquantity =
ec.EXPCss[intSupSched].SDSHagent[intSeller].intAGSHno_units;
```

```
sellers[intSeller].blnAGNTactive = true;
sellers[intSeller].dblAGNTactualgain = 0.0;
// (note: only allows for one limit price)
sellers[intSeller].dblAGNTlimit =
ec.EXPCss[intSupSched].SDSHagent[intSeller].dblAGSHlimit[0];
// SET THE PRICE OF THE AGENT FROM ITS LIMIT AND PROFIT VALUES
sellers[intSeller].AGNT_calcPrice();
if ( blnOutput )
{
psOUT.println( "ZIP_SIM.init_day: Seller " + intSeller
+ " price " + sellers[intSeller].dblAGNTprice );
}
}
if ( blnOutput )
{
psOUT.println();
}
// ------------------------------------------------------------
// IDENTIFY THEORETICAL EQUILIBRIUM PRICE
sdvis.SDVIS_sup_dem( intNoSellers, sellers, intNoBuyers, buyers,
ec.intEXPCmax_trades, dblPrice_0, intQ_0, dblMaxSurplus,
EQ_THEORY, 0.0, blnOutput, psSDO );
dblPrice_0 = sdvis.dblEqPrice;
intQ_0 = sdvis.intIQuant;
dblMaxSurplus = sdvis.dblSurplus;
// SET THEORETICAL GAINS FOR BUYERS AND SELLERS
for ( int intBuyer = 0; intBuyer < intNoBuyers; intBuyer++ )
{
dblEqProfit = buyers[intBuyer].dblAGNTquantity *
( buyers[intBuyer].dblAGNTlimit - dblPrice_0 );
if ( dblEqProfit < 0.0 )
{
dblEqProfit = 0.0;
}
buyers[intBuyer].dblAGNTtheorgain = dblEqProfit;
}
for ( int intSeller = 0; intSeller < intNoSellers; intSeller++ )
{
dblEqProfit = sellers[intSeller].dblAGNTquantity *
( dblPrice_0 - sellers[intSeller].dblAGNTlimit );
if ( dblEqProfit < 0.0 )
{
dblEqProfit = 0.0;
}
sellers[intSeller].dblAGNTtheorgain = dblEqProfit;
}
}
// ------------------------------------------------------------------
// SIM_TRADE: see if a buyer and seller can be found who will enter
// into a trade (called by ZIP_Sim.SIM_main)
public void SIM_trade( ZIP_Data_Trade data_trade, ZIP_Agent[] sellers,
ZIP_Agent[] buyers, ZIP_Exp_Control ec, boolean blnOutput,
PrintStream psOUT, ZIP_SD_Vis sdvis, Random random )
{
// index of buyer randomly selected to be able to trade
int intBuyer = 0;
// index of seller randomly selected to be able to trade
int intSeller = 0;
// deal type
boolean blnDealType;
// status (deal or no deal?)
// (a global variable)
// equilibrium quantity
int intEqQuantity = 0;
// no. of agents willing to trade at a given price
int intNoWilling = 0;
// number of agents able to trade at a given price
int intNoAble;
// number of failed / declined bids / offers
```

```
int intNoFails;
// number of buyers
int intNoBuyers;
// number of sellers
int intNoSellers;
// number of active buyers
int intNoBuyActive;
// number of active sellers
int intNoSellActive;
// can buyer shout offers?
boolean blnBuyShout;
// can sellers shout offers?
boolean blnSellShout;
// number of traders to choose from when generating a shout
int intTraders;
// flag raised until an opening offer is made
boolean blnFirstOffer;
// flag raised until an opening bid is made
boolean blnFirstBid;
// list of agent indices
int[] intAgentList = new int[MAX_AGENTS];
// equilibrium price
double dblEqPrice = 0.0;
// current actual maximum surplus
double dblCurrMaxSurp = 0.0;
// used in NYSE rules
double dblBestOffer = 0.0;
// used in NYSE rules
double dblBestBid = 0.0;
// price of bid / offer
double dblPrice = 0.0;
// get number of buyer and seller agents
intNoBuyers = ec.EXPCds[ec.intEXPCds].intSDSHno_agents;
intNoSellers = ec.EXPCss[ec.intEXPCss].intSDSHno_agents;
// get whether they can shout
blnBuyShout = ec.EXPCds[ec.intEXPCds].blnSDSHcan_shout;
blnSellShout = ec.EXPCss[ec.intEXPCss].blnSDSHcan_shout;
if (( blnBuyShout == false ) && ( blnSellShout == false ))
{
psOUT.println(
"FAIL: Can't have both buyers AND sellers silent" );
System.out.println(
"FAIL: Can't have both buyers AND sellers silent" );
System.exit( 0 );
}
// FIND THE THEORETICAL EQUILIBRIUM PRICE
sdvis.SDVIS_sup_dem( intNoSellers, sellers, intNoBuyers, buyers,
ec.intEXPCmax_trades, dblEqPrice, intEqQuantity, dblCurrMaxSurp,
EQ_THEORY, 0.0, blnOutput, psSDO );
dblEqPrice = sdvis.dblEqPrice;
intEqQuantity = sdvis.intIQuant;
dblCurrMaxSurp = sdvis.dblSurplus;
// UPDATE TRADE STATISTICS
if ( dblEqPrice != NULL_EQ )
{
data_trade.dblTDATtheor_eq_price = dblEqPrice;
data_trade.intTDATtheor_eq_quant = intEqQuantity;
}
else
{
data_trade.intTDATtheor_eq_quant = NULL_EQ;
}
// FIND THE ACTUAL EQUILIBRIUM PRICE
sdvis.SDVIS_sup_dem( intNoSellers, sellers, intNoBuyers, buyers,
ec.intEXPCmax_trades, dblEqPrice, intEqQuantity, dblCurrMaxSurp,
EQ_ACTUAL, 0.0, blnOutput, psSDO );
dblEqPrice = sdvis.dblEqPrice;
intEqQuantity = sdvis.intIQuant;
dblCurrMaxSurp = sdvis.dblSurplus;
// UPDATE TRADE STATISTICS
```

```
if ( dblEqPrice != NULL_EQ )
{
data_trade.dblTDAtactual_eq_price = dblEqPrice;
data_trade.intTDAtactual_eq_quant = intEqQuantity;
}
else
{
data_trade.dblTDAtactual_eq_price = NULL_EQ;
}
intNoFails = 0;
intStatus = NO_DEAL;
blnFirstOffer = true;
blnFirstBid = true;
// -------------------------------------------------------------
// WHILE NO DEAL
while (( intStatus == NO_DEAL ) && ( intNoFails < MAX_FAILS ))
{
// count active agents and mark them as able (i.e. allowed
// to trade at this price) to bid (buyers) or offer (sellers)
intNoBuyActive = 0;
for ( intBuyer = 0; intBuyer < intNoBuyers; intBuyer++ )
{
if ( buyers[intBuyer].blnAGNTactive )
{
buyers[intBuyer].blnAGNTable = true;
intNoBuyActive++;
}
else
{
buyers[intBuyer].blnAGNTable = false;
}
}
intNoSellActive = 0;
for ( intSeller = 0; intSeller < intNoSellers; intSeller++ )
{
if ( sellers[intSeller].blnAGNTactive )
{
sellers[intSeller].blnAGNTable = true;
intNoSellActive++;
}
else
{
sellers[intSeller].blnAGNTable = false;
}
}
intTraders = 0;
if ( blnSellShout )
{
intTraders += intNoSellActive;
}
if ( blnBuyShout )
{
intTraders += intNoBuyActive;
}
if ( blnOutput )
{
// output number of buyers and sellers active
psOUT.println( "ZIP_SIM.sim_trade: " + intTraders
+ " traders, " + intNoSellActive + " SELLERS active, "
+  intNoBuyActive + " BUYERS active" );
psOUT.println();
}
// -------------------------------------------------------------
// SELECT A SELLER TO OFFER (randomly determined)
if ( (int) ( random.nextDouble() * intTraders ) <
intNoSellActive )
{
// is there a seller able to make an offer?
blnDealType = OFFER;
// if using nyse rule and not first offer
if (( ec.blnEXPCnyse ) && ( blnFirstOffer == false ))
{
```

```
// if ZI traders
if ( ec.blnEXPCrandom )
{
// any seller with a limit price higher than best offer
// can't deal
for ( intSeller = 0; intSeller < intNoSellers;
intSeller++ )
{
if ( sellers[intSeller].dblAGNTlimit > dblBestOffer )
{
sellers[intSeller].blnAGNTable = false;
}
}
}
// if ZIP traders
else
{
// any seller with an equal or higher price can't offer
for ( intSeller = 0; intSeller < intNoSellers;
intSeller++ )
{
if ( sellers[intSeller].dblAGNTprice >= dblBestOffer )
{
sellers[intSeller].blnAGNTable = false;
}
}
}
}
// form a list of seller agents able to deal
intNoAble = SIM_get_able( 0.0, sellers, intNoSellers,
intAgentList, blnOutput, psOUT );
if ( blnOutput )
{
psOUT.println();
}
// if there are any sellers able to trade
if ( intNoAble > 0 )
{
// RANDOMLY SELECT A SELLER AGENT ABLE TO TRADE
int intTemp = (int) ( random.nextDouble() * intNoAble );
intSeller = intAgentList[intTemp];
// get the offer price for the seller
dblPrice = SIM_get_price( sellers[intSeller], intSeller,
ec.blnEXPCrandom, blnOutput, psOUT, random );
// if using nyse rule
if ( ec.blnEXPCnyse )
{
// make seller offer the (best) offer
if ( blnFirstOffer )
{
dblBestOffer = dblPrice;
blnFirstOffer = false;
}
else
{
// (nyse rule for bid price improvement)
if ( dblPrice < dblBestOffer )
{
dblBestOffer = dblPrice;
}
}
}
if ( blnOutput )
{
psOUT.println();
}
// get willing buyers
intNoWilling = SIM_get_willing( dblPrice, buyers,
intNoBuyers, intAgentList, ec.blnEXPCrandom, blnOutput,
psOUT, random );
if ( blnOutput )
```

```
{
psOUT.println();
}
// if there are any willing buyers then enter into a DEAL
if ( intNoWilling > 0 )
{
intStatus = DEAL;
}
}
else
{
if ( blnOutput )
{
psOUT.println(
"ZIP_SIM.sim_trade: No sellers able to offer" );
}
intNoFails = MAX_FAILS;
intStatus = END_DAY;
}
}
// ------------------------------------------------------------
// ELSE RANDOMLY SELECT A BUYER AGENT TO BID
else
{
// is there a buyer able to make a bid?
blnDealType = BID;
// if using nyse rule and not first bid
if (( ec.blnEXPCnyse ) && ( blnFirstBid == false ))
{
// if ZI traders
if ( ec.blnEXPCrandom )
{
// any buyer with limit lower than best buyer bid can't
// deal
for ( intBuyer = 0; intBuyer < intNoBuyers; intBuyer++ )
{
if ( buyers[intBuyer].dblAGNTlimit < dblBestBid )
{
buyers[intBuyer].blnAGNTable = false;
}
}
}
else
{
// any buyer with an equal or lower price can't bid
for ( intBuyer = 0; intBuyer < intNoBuyers; intBuyer++ )
{
if ( buyers[intBuyer].dblAGNTprice <= dblBestBid )
{
buyers[intBuyer].blnAGNTable = false;
}
}
}
}
// form a list of buyer agents able to deal
intNoAble = SIM_get_able( 0.0, buyers, intNoBuyers,
intAgentList, blnOutput, psOUT );
if ( blnOutput )
{
psOUT.println();
}
// if there are any buyer agents able to trade at this price
// in the market
if ( intNoAble > 0 )
{
// randomly select a buyer able to trade
int intTemp = (int) ( random.nextDouble() * intNoAble );
intBuyer = intAgentList[intTemp];
// get the bid price price for the buyer
dblPrice = SIM_get_price( buyers[intBuyer], intBuyer,
ec.blnEXPCrandom, blnOutput, psOUT, random );
```

```
// if using nyse rule
if ( ec.blnEXPCnyse )
{
// make buyer bid the (best) bid
if ( blnFirstBid )
{
dblBestBid = dblPrice;
blnFirstBid = false;
}
else
{
// (nyse rule for bid price improvement)
if ( dblPrice > dblBestBid )
{
dblBestBid = dblPrice;
}
}
}
if ( blnOutput )
{
psOUT.println();
}
// get willing sellers
intNoWilling = SIM_get_willing( dblPrice, sellers,
intNoSellers, intAgentList, ec.blnEXPCrandom, blnOutput,
psOUT, random );
if ( blnOutput )
{
psOUT.println();
}
// if there are any willing sellers then enter into a DEAL
if ( intNoWilling > 0 )
{
intStatus = DEAL;
}
}
else
{
if ( blnOutput )
{
psOUT.println(
"ZIP_SIM.sim_trade: No buyers able to bid" );
}
intNoFails = MAX_FAILS;
intStatus = END_DAY;
}
}
// ------------------------------------------------------------
// IF RANDOMLY SELECTED BUYER OR SELLER AGENT ABLE TO TRADE
if ( intStatus == DEAL )
{
// if deal based on a SELLER's offer
if ( blnDealType == OFFER )
{
// randomly select a willing buyer for this offer
int intTemp = (int) ( random.nextDouble() * intNoWilling );
intBuyer = intAgentList[intTemp];
if ( blnOutput )
{
psOUT.println( "ZIP_SIM.sim_trade: SELLER " +
intSeller + " SELLS TO BUYER " + intBuyer +
" (reward = " + SIM_reward( buyers[intBuyer],
dblPrice ) + ")" );
}
}
// if deal based on a BUYER's bid
else
{
// randomly select a willing seller for this bid
int intTemp = (int) ( random.nextDouble() * intNoWilling );
```

```
intSeller = intAgentList[intTemp];
if ( blnOutput )
{
psOUT.println( "ZIP_SIM.sim_trade: BUYER " + intBuyer +
" BUYS FROM SELLER " + intSeller + " (reward = " +
SIM_reward( sellers[intSeller], dblPrice ) + ")" );
}
}
// UPDATE TRADE DATA: record trade price and deal type
data_trade.dblTDATdeal_price = dblPrice;
data_trade.blnTDATdeal_type = blnDealType;
if ( blnOutput )
{
psOUT.println();
}
// now a trade has taken place, update the trading strategies
// of buyers
for ( int intBuyerAg = 0; intBuyerAg < intNoBuyers;
intBuyerAg++ )
{
buyers[intBuyerAg].AGNT_shout_update_buyer( intBuyerAg,
blnDealType, intStatus, dblPrice, blnOutput, psOUT,
random );
}
if ( blnOutput )
{
psOUT.println();
}
// now a trade has taken place, update the trading strategies
// of sellers
for ( int intSellerAg = 0; intSellerAg < intNoSellers;
intSellerAg++ )
{
sellers[intSellerAg].AGNT_shout_update_seller( intSellerAg,
blnDealType, intStatus, dblPrice, blnOutput, psOUT,
random );
}
// update the bank accounts of buyer and seller
SIM_bank( sellers[intSeller], buyers[intBuyer], dblPrice,
blnOutput, psOUT );
if ( blnOutput )
{
psOUT.println();
}
}
// ------------------------------------------------------------
// IF RANDOMLY SELECTED BUYER OR SELLER AGENT NOT ABLE TO TRADE
else
{
// NO DEAL or END DAY
// increment number of fails count (if not already set at max
// fails)
if ( intNoFails < MAX_FAILS )
{
intNoFails++;
}
if ( blnOutput )
{
psOUT.println(
"ZIP_SIM.sim_trade: No willing takers (fails = "
+ intNoFails + ")" );
psOUT.println();
}
// (negative price in trade data = no deal)
data_trade.dblTDATdeal_price = -1.0;
// update the trading strategies of buyers
for ( int intBuyerAg = 0; intBuyerAg < intNoBuyers;
intBuyerAg++ )
{
```

```
buyers[intBuyerAg].AGNT_shout_update_buyer( intBuyerAg,
blnDealType, intStatus, dblPrice, blnOutput, psOUT,
random );
}
// update the trading strategies of sellers
for ( int intSellerAg = 0; intSellerAg < intNoSellers;
intSellerAg++ )
{
sellers[intSellerAg].AGNT_shout_update_seller(
intSellerAg, blnDealType, intStatus, dblPrice,
blnOutput, psOUT, random );
}
}
}
}
// ---------------------------------------------------------------
// SIM_REWARD: given a bid or offer price, returns the monetary reward
// for a deal (called by ZIP_Sim.SIM_trade, SIM_get_price,
// SIM_get_willing, SIM_get_able and SIM_bank)
public double SIM_reward( ZIP_Agent agent, double dblAGNTprice )
{
double dblReward = 0.0;
// if a SELLER agent
if ( agent.blnAGNTtype == SELL )
{
// reward = offer price - agent limit price
dblReward = ( dblAGNTprice - agent.dblAGNTlimit );
}
// if a BUYER agent
if ( agent.blnAGNTtype == BUY )
{
// reward = agent limit price - bid price
dblReward = ( agent.dblAGNTlimit - dblAGNTprice );
}
// ensure any negative rewards set at 0.0
if ( dblReward < 0.0 )
{
dblReward = 0.0;
}
return dblReward;
}
// ---------------------------------------------------------------
// SIM_GET_PRICE: get a (bid or offer) price for an agent
// (called by ZIP_Sim.SIM_trade, SIM_get_willing)
public double SIM_get_price( ZIP_Agent agent, int intID,
boolean blnRandom, boolean blnOutput, PrintStream psOUT,
Random random )
{
double dblPrice;
// bounds on random prices
double dblRandomMin = 0.01;
double dblRandomMax = 4.0;
// if ZI trader
if ( blnRandom )
{
// maximum bound for random price must not be less than agent
// limit price
if ( dblRandomMax < agent.dblAGNTlimit )
{
psOUT.println(
"FAIL: dblRandomMax too low in AGNT_get_price" );
System.out.println(
"FAIL: dblRandomMax too low in AGNT_get_price" );
System.exit( 0 );
}
// if a BUYER agent
if ( agent.blnAGNTtype == BUY )
{
// calculate bid price randomly (within bounds)
```

```
dblPrice = dblRandomMin + (( random.nextDouble() / 10 ) *
( agent.dblAGNTlimit - dblRandomMin ) );
}
// else a SELLER agent
else
{
// calculate offer price randomly (within bounds)
dblPrice = agent.dblAGNTlimit + (( random.nextDouble() / 10 )
 * ( dblRandomMax - agent.dblAGNTlimit ) );
}
// adjust price
dblPrice = ( Math.floor( 0.5 + ( dblPrice * 100 ) ) ) / 100;
agent.dblAGNTprice = dblPrice;
}
// else if a ZIP trader
else
{
dblPrice = agent.dblAGNTprice;
}
// output to log bid or offer price
if ( blnOutput )
{
// get monetary reward for the deal
double dblReward = SIM_reward( agent, dblPrice );
if ( agent.blnAGNTtype == BUY )
{
psOUT.println( "ZIP_SIM: BUYER " + intID + " BIDS AT "
+ dblPrice + " (reward = " + dblReward + ")" );
}
else
{
psOUT.println( "ZIP_SIM: SELLER " + intID + " OFFERS AT "
+ dblPrice + " (reward = " + dblReward + ")" );
}
}
return dblPrice;
}
// ---------------------------------------------------------------------
// SIM_GET_WILLING: form a list of agents willing to deal
// (called by ZIP_Sim.SIM_trade)
public int SIM_get_willing( double dblOtherPrice, ZIP_Agent agent[],
int intNoAgents, int intAgentList[], boolean blnRandom,
boolean blnOutput, PrintStream psOUT, Random random )
{
int intWilling = 0;
double dblThisPrice;
double dblPrice = dblOtherPrice;
for ( int intAgent = 0; intAgent < intNoAgents; intAgent++ )
{
// if ZI trader
if ( blnRandom )
{
// (agent generates a price at random and compares it to a
// given price and is willing IF random price makes a profit)
// set willing to trade at this price flag initially to false
agent[intAgent].blnAGNTwilling = false;
if ( agent[intAgent].blnAGNTactive )
{
// get the (bid or offer) price for the current agent
dblThisPrice = SIM_get_price( agent[intAgent], intAgent,
blnRandom, blnOutput, psOUT, random );
// if a BUYER
if ( agent[intAgent].blnAGNTtype == BUY )
{
// if agents bid price is larger than the sellers offer
// price then set as willing
if ( dblThisPrice > dblOtherPrice )
{
agent[intAgent].blnAGNTwilling = true;
dblPrice = dblThisPrice;
```

```
}
}
// if a SELLER
else
{
// if agents offer price is less than the buyers bid
// price then set as willing
if ( dblThisPrice < dblOtherPrice )
{
agent[intAgent].blnAGNTwilling = true;
dblPrice = dblThisPrice;
}
}
}
}
// else if ZIP trader
else
{
// use some intelligence to determine whether willing to
// trade at this price or not
agent[intAgent].AGNT_willing_trade( dblPrice );
}
// if agent willing
if ( agent[intAgent].blnAGNTwilling )
{
// add to the willing list and add to total number of willing
// agents
intAgentList[intWilling] = intAgent;
intWilling++;
// get monetary reward for the deal
double dblReward = SIM_reward( agent[intAgent], dblPrice );
if ( blnOutput )
{
if ( agent[intAgent].blnAGNTtype == BUY )
{
psOUT.println( "ZIP_SIM: BUYER " + intAgent +
" willing (reserve) price = " + dblPrice
+ " (reward = " + dblReward + ")" );
}
else
{
psOUT.println( "ZIP_SIM: SELLER " + intAgent +
" willing (reserve) price = " + dblPrice  +
" (reward = " + dblReward + ")" );
}
}
}
}
if ( blnOutput )
{
psOUT.println();
psOUT.println( "ZIP_SIM: " + intWilling +
" traders willing to deal" );
}
return intWilling;
}
// ---------------------------------------------------------------
// SIM_GET_ABLE: form a list of agents able to deal
// (called by ZIP_Sim.SIM_trade)
public int SIM_get_able( double dblPrice, ZIP_Agent agent[],
int intNoAgents, int intAgentList[], boolean blnOutput,
PrintStream psOUT )
{
int intAble = 0;
for ( int intAgent = 0; intAgent < intNoAgents; intAgent++ )
{
// if agent able to trade
if ( agent[intAgent].blnAGNTable )
{
// add agent index to the list of agents and add to total
```

```
// number of able agents
intAgentList[intAble] = intAgent;
intAble++;
// get the monetary reward for the deal
double dblReward = SIM_reward( agent[intAgent], dblPrice );
if ( blnOutput )
{
if ( agent[intAgent].blnAGNTtype == BUY )
{
psOUT.println( "ZIP_SIM.SIM_get_able: BUYER "
+ intAgent + " able (reward = " + dblReward + ")" );
}
else
{
psOUT.println( "ZIP_SIM.SIM_get_able: SELLER "
+ intAgent + " able (reward = " + dblReward + ")" );
}
}
}
}
return intAble;
}
// ---------------------------------------------------------------
// SIM_BANK: adjust bank balances of buyer and seller agents involved
// in a deal (called by ZIP_Sim.SIM_trade)
public void SIM_bank( ZIP_Agent seller, ZIP_Agent buyer,
double dblPrice, boolean blnOutput, PrintStream psOUT )
{
double dblReward;
// SELLER
// get monetary reward for the deal
dblReward = SIM_reward( seller, dblPrice );
// add monetary reward for the deal to the agents' bank account
seller.dblAGNTmoney += dblReward;
seller.dblAGNTactualgain += dblReward;
dblSurplus += dblReward;
// decrease by one the total number of commodities the agent has
seller.dblAGNTquantity--;
// if the agent has no quantity of the commodity then set as
// inactive
if ( seller.dblAGNTquantity < 1 )
{
seller.blnAGNTactive = false;
}
if ( blnOutput )
{
psOUT.println();
psOUT.println( "ZIP_SIM.SIM_bank: SELLER limit = "
+ seller.dblAGNTlimit + ", reward = " + dblReward +
", money = " + seller.dblAGNTmoney + ", quantity = " +
seller.dblAGNTquantity + " (surplus = " + dblSurplus
+ " )" );
}
// BUYER
// get monetary reward for the deal
dblReward = SIM_reward( buyer, dblPrice );
// add monetary reward for the deal to the agents' bank account
buyer.dblAGNTmoney += dblReward;
buyer.dblAGNTactualgain += dblReward;
dblSurplus += dblReward;
// decrease by one the total number of commodities the agent has
buyer.dblAGNTquantity--;
// if the agent has no quantity of the commodity then set as
// inactive
if ( buyer.dblAGNTquantity < 1 )
{
buyer.blnAGNTactive = false;
}
if ( blnOutput )
```

```
{
psOUT.println();
psOUT.println( "ZIP_SIM.SIM_bank: BUYER limit = "
+ buyer.dblAGNTlimit + ", reward = " + dblReward +
", money = " + buyer.dblAGNTmoney + ", quantity = " +
buyer.dblAGNTquantity + " (surplus = " + dblSurplus + " )" );
}
}
// ---------------------------------------------------------------
}
```

C.2 MBC Simulation Classes

As for the ZIP simulation, the MBC simulation is decomposed into a number of well-defined classes and data structures. The MBC Simulation also uses the ZIP_Agent, ZIP_Constants, ZIP_Data_Day, ZIP_Data_Trade and ZIP_SD_Vis classes.

C.2.1 MBC_GA.java

Description: The GA used to evolve MBC-based ZIP trading agent parameter sets and marketplaces.

```
// Class: MBC_GA.java
// ALGORITHM:
// fill population with random values
// for n times round generation loop
//  evaluate fitness of all genomes in the population
//  select preferentially the fitter ones as parents
//  for <intPopSize> times round repro loop
//  pick 2 from parental pool
// recombine to make 1 offspring
// mutate the offspring
//  end repro loop
// throw away parental generation and replace with offspring
// end generation loop
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.text.DecimalFormat;
import java.util.Random;
import java.util.Hashtable;
public class MBC_GA
{
// number of generations
private int intNoOfGens = 100;
// number of MBC trials
private int intNoTrials = 20;
// population size
private int intPopSize = 30;
// genome contains 17 parameters
private int intGenSize = 17;
PrintStream psDAY, psTRA, psSDO, psGA, psGEN;
private UDC_Network UDCnet;
public void GA( int intGARun )
{
// number of ZIP_Sim experiments to carry out for each individual
int intNoTimeSteps = 125;
Random random = new Random();
//random.setSeed( 500 );
String strFileName1;
String strFileName2;
// ------------------------------------------------------------------
double dblPop[][] = new double[intGenSize][intPopSize];
double dblNewPop[][] = new double[intGenSize][intPopSize];
double dblFitnessScore[] = new double[intPopSize];
double dblFittestGen[] = new double[intGenSize];
double dblCrossoverProb = 0.125;
double dblMutationProb = 0.3;
int intGen = 1;
try
{
```

```
psGA = new PrintStream( new FileOutputStream(
"ZIP_GA_EASY_ELITE_GEN_exp" + intGARun + ".txt" ) );
psGEN = new PrintStream( new FileOutputStream(
"ZIP_GA_EASY_POP_SCORES_exp" + intGARun + ".txt" ) );
}
catch( Exception exception )
{
System.out.println( "IOException: " + exception );
System.exit( 0 );
}
// header for 'elite genomes' output file
psGA.println(
"LR_MIN LR_MAX MM_MIN MM_MAX PR_MIN PR_MAX TR_R TR_A "
+ "LR_MIN LR_MAX MM_MIN MM_MAX PR_MIN PR_MAX TR_R "
+ "TR_A QS GEN BEST FITNESS" );
try
{
// initialise file to write timestep data to
psDAY = new PrintStream( new FileOutputStream( "MBC_DAY_exp"
+ intGARun + ".txt" ) );
// initialise file to write trade data to
psTRA = new PrintStream( new FileOutputStream( "MBC_TRA_exp"
+ intGARun + ".txt" ) );
}
catch( Exception exception )
{
System.out.println( "IOException: " + exception );
System.exit( 0 );
}
// ---------------------------------------------------------------------
// BUILD UDC_NETWORK
MBC_Sim sim = new MBC_Sim();
UDCnet = sim.SIM_build_udc( random, false );
final Hashtable UDC = UDCnet.getNetwork();
// ---------------------------------------------------------------------
// FILL POPULATION WITH INITIAL PARAMETER VALUES
// whole population initialised with same parameter values
for ( int intPopPos = 0; intPopPos < intPopSize; intPopPos++ )
{
for ( int intGenPos = 0; intGenPos < intGenSize; intGenPos++ )
{
dblPop[0][intPopPos] = 0.10;
dblPop[1][intPopPos] = 0.40;
dblPop[2][intPopPos] = 0.00;
dblPop[3][intPopPos] = 0.10;
dblPop[4][intPopPos] = 0.05;
dblPop[5][intPopPos] = 0.30;
dblPop[6][intPopPos] = 0.05;
dblPop[7][intPopPos] = 0.05;
dblPop[8][intPopPos] = 0.10;
dblPop[9][intPopPos] = 0.40;
dblPop[10][intPopPos] = 0.00;
dblPop[11][intPopPos] = 0.10;
dblPop[12][intPopPos] = 0.05;
dblPop[13][intPopPos] = 0.30;
dblPop[14][intPopPos] = 0.05;
dblPop[15][intPopPos] = 0.05;
dblPop[16][intPopPos] = 0.5;
}
}
// ---------------------------------------------------------------------
// GENERATIONAL LOOP
do
{
System.out.print( "RUN:" + intGARun + " GEN:" + intGen );
// -----------------------------------------------------------------
// EVALUATE FITNESS OF GENOMES IN POPULATION
dblFitnessScore = fitness_function( dblPop, dblFitnessScore,
intNoTimeSteps, random, intGen, intGARun, intNoOfGens, UDCnet );
```

```
// ----------------------------------------------------------------
// (OUTPUT GA GENERATION DATA)
// (set to a high value as minimising)
double dblFittestScore = 100000.0;
// identify fittest genome (for elitism and output)
for ( int intPopPos = 0; intPopPos < intPopSize; intPopPos++ )
{
// (MINIMISING HERE)
if ( dblFitnessScore[intPopPos] < dblFittestScore )
{
dblFittestScore = dblFitnessScore[intPopPos];
for ( int intGenPos = 0; intGenPos < intGenSize; intGenPos++ )
{
dblFittestGen[intGenPos] = dblPop[intGenPos][intPopPos];
}
}
}
DecimalFormat df = new DecimalFormat( "0.00000" );
// output elite genome
for ( int intGenPos = 0; intGenPos < intGenSize; intGenPos++ )
{
psGA.print( df.format( dblFittestGen[intGenPos] ) + " " );
}
psGA.println( intGen + " " + dblFittestScore );
// output population
for ( int intPopPos = 0; intPopPos < intPopSize; intPopPos++ )
{
psGEN.print( dblFitnessScore[intPopPos] + " " );
}
psGEN.println();
// ----------------------------------------------------------------
// SELECT PREFERENTIALLY FITTER GENOMES AS PARENTS
double[] dblGenome1 = new double[intGenSize];
double[] dblGenome2 = new double[intGenSize];
double[] dblGenome3 = new double[intGenSize];
// ----------------------------------------------------------------
// REPRODUCTION LOOP (for intPopSize times)
// create a new population
for ( int intPopPos = 0; intPopPos < intPopSize; intPopPos++ )
{
// randomly select three (unique) genomes from existing
// population
// generate a first random genome selection
int intSelectedGen1 = random.nextInt( intPopSize );
int intSelectedGen2 = intSelectedGen1;
// generate a second random genome selection not the same as the
// first
while ( intSelectedGen2 == intSelectedGen1 )
{
intSelectedGen2 = random.nextInt( intPopSize );
}
int intSelectedGen3 = intSelectedGen1;
// generate a third random genome selection not the same as the
// first two
while ( intSelectedGen3 == intSelectedGen1 )
{
intSelectedGen3 = intSelectedGen2;
while ( intSelectedGen3 == intSelectedGen2 )
{
intSelectedGen3 = random.nextInt( intPopSize );
}
}
// get the randomly selected genomes from the population
for ( int intGenPos = 0; intGenPos < intGenSize; intGenPos++ )
{
dblGenome1[intGenPos] = dblPop[intGenPos][intSelectedGen1];
dblGenome2[intGenPos] = dblPop[intGenPos][intSelectedGen2];
dblGenome3[intGenPos] = dblPop[intGenPos][intSelectedGen3];
```

```
}
// identify the fittest two (become parents) of these three
// (discard most unfit)
// fittest parent genome becomes Vmom, other parent genome Vdad
double dblFitness1 = dblFitnessScore[intSelectedGen1];
double dblFitness2 = dblFitnessScore[intSelectedGen2];
double dblFitness3 = dblFitnessScore[intSelectedGen3];
double dblMom = 100000.0;
double dblDad = 100000.0;
if ( dblFitness1 < dblMom )
{
dblMom = dblFitness1;
if ( dblFitness2 < dblMom )
{
dblMom = dblFitness2;
}
if ( dblFitness3 < dblMom )
{
dblMom = dblFitness3;
}
}
int intMaPos = 0;
int intPaPos = 0;
// if dblFitness1 was the fittest
if ( dblFitness1 == dblMom )
{
intMaPos = intSelectedGen1;
// identify fittest between dblFitness2 and dblFitness3
if ( dblFitness2 < dblFitness3 )
{
dblDad = dblFitness2;
intPaPos = intSelectedGen2;
}
else
{
dblDad = dblFitness3;
intPaPos = intSelectedGen3;
}
}
// if dblFitness2 was the fittest
else if ( dblFitness2 == dblMom )
{
intMaPos = intSelectedGen2;
// identify fittest between dblFitness1 and dblFitness3
if ( dblFitness1 < dblFitness3 )
{
dblDad = dblFitness1;
intPaPos = intSelectedGen1;
}
else
{
dblDad = dblFitness3;
intPaPos = intSelectedGen3;
}
}
// if dblFitness3 was the fittest
else
{
intMaPos = intSelectedGen3;
// identify fittest between dblFitness1 and dblFitness2
if ( dblFitness1 < dblFitness2 )
{
dblDad = dblFitness1;
intPaPos = intSelectedGen1;
}
else
{
dblDad = dblFitness2;
intPaPos = intSelectedGen2;
}
}
// CROSSOVER: recombine to make 1 offspring
// begin by copying the first element of the Ma genome into
// new Pop
dblNewPop[0][intPopPos] = dblPop[0][intMaPos];
```

```
boolean blnParent = true;
// uniform recombination of ma and pa genomes
// (with probability dblCrossoverProb)
for ( int intGenPos = 1; intGenPos < intGenSize; intGenPos++ )
{
if ( random.nextDouble() < dblCrossoverProb )
{
// change parent
if ( blnParent == true )
{
dblNewPop[intGenPos][intPopPos] =
dblPop[intGenPos][intPaPos];
blnParent = false;
}
else
{
dblNewPop[intGenPos][intPopPos] =
dblPop[intGenPos][intMaPos];
blnParent = true;
}
}
else
{
// stay with same parent
if ( blnParent == true )
{
dblNewPop[intGenPos][intPopPos] =
dblPop[intGenPos][intMaPos];
}
else
{
dblNewPop[intGenPos][intPopPos] =
dblPop[intGenPos][intPaPos];
}
}
}
// MUTATION: mutate the offspring
for ( int intGenPos = 0; intGenPos < intGenSize; intGenPos++ )
{
// (mutation always carried out)
// generate a random number in range [-0.05,0.05]
double dblRandMut = ( random.nextDouble() * 0.10 ) - 0.05;
dblNewPop[intGenPos][intPopPos] += dblRandMut;
// clip to ensure value in range [0,1]
if ( dblNewPop[intGenPos][intPopPos] > 1.0 )
{
dblNewPop[intGenPos][intPopPos] = 1.0;
}
if ( dblNewPop[intGenPos][intPopPos] < 0.0 )
{
dblNewPop[intGenPos][intPopPos] = 0.0;
}
}
}
// END REPRODUCTION LOOP
// ----------------------------------------------------------------
// ELITISM: retain best genome of this generation (put into first
// slot of new population)
for ( int intGenPos = 0; intGenPos < intGenSize; intGenPos++ )
{
dblNewPop[intGenPos][0] = dblFittestGen[intGenPos];
}
// throw away parental generation and replace with offspring
for ( int intPopPos = 0; intPopPos < intPopSize; intPopPos++ )
{
for ( int intGenPos = 0; intGenPos < intGenSize; intGenPos++ )
{
dblPop[intGenPos][intPopPos] =
dblNewPop[intGenPos][intPopPos];
}
```

```
}
intGen++;
}
// END GENERATION LOOP
while( intGen <= intNoOfGens );
// ------------------------------------------------------------------
}
// ------------------------------------------------------------------
// FITNESS_FUNCTION: return fitness scores
public double[] fitness_function( double dblPop[][],
double dblFitnessScore[], int intNoTimeSteps, Random random,
int intGen, int intGARun, int intNoGens, UDC_Network UDCnet )
{
// iterate through each member of the population
for ( int intPopPos = 0; intPopPos < intPopSize; intPopPos++ )
{
if ( ( ( intGen % 50 ) == 0 ) || ( intGen == 1 ) )
{
psDAY.println();
psDAY.println( "----------------------------------------"
+ "---" );
psDAY.println( "MBC_SIM TRIAL STATS, GEN: " + intGen + ", "
+ "POP MEMBER: " + intPopPos );
psDAY.println( "----------------------------------------"
+ "---" );
psDAY.println();
psTRA.println();
psTRA.println( "----------------------------------------"
+ "---" );
psTRA.println( "MBC_SIM TRADE STATS, GEN: " + intGen + ", "
+ "POP MEMBER: " + intPopPos );
psTRA.println( "----------------------------------------"
+ "---" );
psTRA.println();
}
for ( int intTrial = 0; intTrial < intNoTrials; intTrial++ )
{
// call main function
MBC_Sim sim = new MBC_Sim();
// 'reset' udc network
sim.reset_UDC( UDCnet, intNoTimeSteps );
// run MBC trial
dblFitnessScore[intPopPos] += sim.SIM_main(
intNoTimeSteps, dblPop[0][intPopPos],
dblPop[1][intPopPos], dblPop[2][intPopPos],
dblPop[3][intPopPos], dblPop[4][intPopPos],
dblPop[5][intPopPos], dblPop[6][intPopPos],
dblPop[7][intPopPos], dblPop[8][intPopPos],
dblPop[9][intPopPos], dblPop[10][intPopPos],
dblPop[11][intPopPos], dblPop[12][intPopPos],
dblPop[13][intPopPos], dblPop[14][intPopPos],
dblPop[15][intPopPos], dblPop[16][intPopPos],
intGen, intPopPos, random, psDAY, psTRA, psSDO,
intNoGens, UDCnet );
}
// output GA progress
if ( ( intPopPos % 10 ) == 0 )
{
System.out.print( " " );
}
System.out.print( "*" );
if ( intPopPos == ( intPopSize - 1 ) )
{
System.out.println();
}
// average fitness score over number of trials
double dblTemp = dblFitnessScore[intPopPos] / intNoTrials;
dblFitnessScore[intPopPos] = dblTemp;
```

```
}
return dblFitnessScore;
}
// ----------------------------------------------------------------
public static void main ( String args[] )
{
MBC_GA ga = new MBC_GA();
int intNoGARuns = 8;
for ( int intGARun = 1; intGARun <= intNoGARuns; intGARun++ )
{
ga.GA( intGARun );
}
}
// ----------------------------------------------------------------
}
```

C.2.2 MBC_Sim.java

Description: Main simulation class. Initialises and initiates updating of the simulation at each timestep.

```
// Class:  MBC_Sim.java
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Random;
import java.util.Vector;
import java.text.DecimalFormat;
public class MBC_Sim implements ZIP_Constants
{
// GLOBAL OBJECTS/PARAMETERS
// flag for UDC network visualisation
private boolean blnSERVvis = true;
// flag for failing network servers
private boolean blnRemoveServers = false;
// timestep to remove servers from network
private int intRemTimeStep = 30;
// trial fitness value
private double dblTrialFitness = 0.0;
// equilibrium price calculated in initialisation method
private double dblPrice_0;
private double dblMaxSurplus;
// number of market auctions taken place
private double dblAuctions = 0.0;
// number of successful market auctions taken place
private double dblSuccAucs = 0.0;
// current simulation timestep
private int intTimeStep = 0;
private int[] intTrades;
// count of total number of tasks processed by the UDC at end of
// simulation
private int intTotalJobsPrsd = 0;
// count of total number of tasks allocated by the UDC at end of
// simulation
private int intTotalJobsAlloc = 0;
// percentage of tasks allocated to total number of tasks put onto
// network
private double dblTaskAllocPerc = 0.0;
private PrintStream psOUT;
// UDC_Vis DECLARATIONS
// number of servers in the UDC_Network
private int intVISno_servers = 100;
// stores each servers load at each timestep
private double[][] dblVISload;
// stores the number of tasks at each servers local market at each
// timestep
private double[][] dblVISno_tasks;
// stores the quote price of each server seller agent at each timestep
private double[][] dblVISpriceSS;
// stores the quote price of each server buyer agent at each timestep
private double[][] dblVISpriceSB;
// stores the number of tasks finished by each server up to each
// timestep
private double[][] dblVISjprsd;
// stores the number of tasks that have been allocated onto network
private double[][] dblVISalloc;
// stores percentage of tasks allocated over all tasks on network
private double[][] dblVISperf;
boolean blnTrade[];
// ---------------------------------------------------------------------
// SIM_MAIN
public double SIM_main( int intVISno_timesteps,
```

```
double BUYdblAGNT_lr_L, double BUYdblAGNT_lr_H,
double BUYdblAGNT_mm_L, double BUYdblAGNT_mm_H,
double BUYdblAGNT_pf_L, double BUYdblAGNT_pf_H,
double BUYdblAGNT_tr_R, double BUYdblAGNT_tr_A,
double SELdblAGNT_lr_L, double SELdblAGNT_lr_H,
double SELdblAGNT_mm_L, double SELdblAGNT_mm_H,
double SELdblAGNT_pf_L, double SELdblAGNT_pf_H,
double SELdblAGNT_tr_R, double SELdblAGNT_tr_A,
double dblQS, int intGen, int intPopPos, Random random,
PrintStream psDAY, PrintStream psTRA, PrintStream psSDO,
int intNoGens, UDC_Network UDCnet )
{
ZIP_Stats stats = new ZIP_Stats();
ZIP_SD_Vis SDVIS = new ZIP_SD_Vis();
boolean blnOutput = false;
intTrades = new int[intVISno_timesteps];
blnTrade = new boolean[intVISno_timesteps];
// INITIALISE OUTPUT STREAMS AND OUTPUT FILES
try
{
psOUT = new PrintStream( new FileOutputStream( "MBC_OUT.txt" ) );
}
catch( Exception exception )
{
System.out.println( "IOException: " + exception );
System.exit( 0 );
}
// ---------------------------------------------------------------------
Hashtable UDC = UDCnet.getNetwork();
// to store all server available capacities for intVisTS timesteps
dblVISload = new double[UDC.size()][intVISno_timesteps];
dblVISno_tasks = new double[UDC.size()][intVISno_timesteps];
dblVISpriceSS = new double[UDC.size()][intVISno_timesteps];
dblVISpriceSB = new double[UDC.size()][intVISno_timesteps];
dblVISjprsd = new double[UDC.size()][intVISno_timesteps];
dblVISalloc = new double[UDC.size()][intVISno_timesteps];
dblVISperf = new double[UDC.size()][intVISno_timesteps];
// ---------------------------------------------------------------------
// initialise server agents
for ( int intServer = 1; intServer <= UDC.size(); intServer++ )
{
UDC_Server server = (UDC_Server) UDC.get( "S" + intServer );
// set QS value on server
server.dblQS = dblQS;
// (GA) SET SELLER AGENT GA PARAMETERS
server.agtSERVseller.dblAGNT_lr_L = SELdblAGNT_lr_L;
server.agtSERVseller.dblAGNT_lr_H = SELdblAGNT_lr_H;
server.agtSERVseller.dblAGNT_mm_L = SELdblAGNT_mm_L;
server.agtSERVseller.dblAGNT_mm_H = SELdblAGNT_mm_H;
server.agtSERVseller.dblAGNT_pf_L = SELdblAGNT_pf_L;
server.agtSERVseller.dblAGNT_pf_H = SELdblAGNT_pf_H;
server.agtSERVseller.dblAGNT_tr_R = SELdblAGNT_tr_R;
server.agtSERVseller.dblAGNT_tr_A = SELdblAGNT_tr_A;
// (GA) SET BUYER AGENT GA PARAMETERS
server.agtSERVbuyer.dblAGNT_lr_L = BUYdblAGNT_lr_L;
server.agtSERVbuyer.dblAGNT_lr_H = BUYdblAGNT_lr_H;
server.agtSERVbuyer.dblAGNT_mm_L = BUYdblAGNT_mm_L;
server.agtSERVbuyer.dblAGNT_mm_H = BUYdblAGNT_mm_H;
server.agtSERVbuyer.dblAGNT_pf_L = BUYdblAGNT_pf_L;
server.agtSERVbuyer.dblAGNT_pf_H = BUYdblAGNT_pf_H;
server.agtSERVbuyer.dblAGNT_tr_R = BUYdblAGNT_tr_R;
server.agtSERVbuyer.dblAGNT_tr_A = BUYdblAGNT_tr_A;
// initialise the buyer and seller agents on the UDC_Server objects
server.agtSERVseller.AGNT_init_seller( intServer,  random,
blnOutput );
server.agtSERVbuyer.AGNT_init_buyer( intServer, random,
blnOutput );
}
// ---------------------------------------------------------------------
```

```
// initialise simulation
SIM_init( blnOutput, psOUT, SDVIS, UDCnet, random, psSDO );
// initialise trade and timestep stat data structures
stats.STATS_init( intVISno_timesteps );
int intTASKid = 0;
for ( int intTimeStep = 1; intTimeStep <= intVISno_timesteps;
intTimeStep++ )
{
// remove server(s) from the network after n timesteps
if ( blnRemoveServers == true )
{
if ( intTimeStep == intRemTimeStep )
{
remove_servers( UDCnet );
}
}
// PASS OVER UDC_NETWORK TO UPDATE SYSTEM
intTASKid = MBC_system_update( intTimeStep, UDCnet, intTrades,
SDVIS, blnOutput, random, BUYdblAGNT_lr_L, BUYdblAGNT_lr_H,
BUYdblAGNT_mm_L, BUYdblAGNT_mm_H, BUYdblAGNT_pf_L,
BUYdblAGNT_pf_H, BUYdblAGNT_tr_R, BUYdblAGNT_tr_A, intTASKid,
stats, psDAY, psTRA, psSDO, psOUT, intVISno_timesteps, intGen,
intNoGens );
}
// return trial fitness score
return stats.dblTrialFitness;
}
// ---------------------------------------------------------------
// MBC_SYSTEM_UPDATE: pass over UDC_Network to update system
// >> called by MBC_Sim.SIM_main
private int MBC_system_update( int intTimeStep, UDC_Network UDCnet,
int[] intTrades, ZIP_SD_Vis SDVIS, boolean blnOutput, Random random,
double dblAGNT_lr_L, double dblAGNT_lr_H, double dblAGNT_mm_L,
double dblAGNT_mm_H, double dblAGNT_pf_L, double dblAGNT_pf_H,
double dblAGNT_tr_R, double dblAGNT_tr_A, int intTASKid,
ZIP_Stats stats, PrintStream psDAY, PrintStream psTRA,
PrintStream psSDO, PrintStream psOUT, int intVISno_timesteps,
int intGen, int intNoGens )
{
blnTrade[intTimeStep-1] = false;
// OUTPUT TIMER
if ( blnOutput )
{
psOUT.println();
psOUT.println( "--------------------------------" );
psOUT.println( "MBC_SIM: Simulation Time Step " + intTimeStep );
psOUT.println( "--------------------------------" );
psOUT.println();
}
// ---------------------------------------------------------------
DecimalFormat format = new DecimalFormat( "0.000" );
// OUTPUT PERCENTAGE OF SUCCESSFUL AUCTIONS
if ( dblAuctions != 0.0 )
{
double dblTemp = ( dblSuccAucs / dblAuctions );
if ( blnOutput )
{
psOUT.println( "MBC_SIM: Percentage of successful auctions "
+ format.format( dblTemp * 100 ) + "% at Timestep "
+ intTimeStep );
psOUT.println();
}
}
intTrades[intTimeStep-1] = 0;
// INITIALISE VARIABLES USED IN CALCULATING TRADE AND TIMESTEP
// STATISTICS (done at the start of every timestep)
stats.STATS_reset();
// ---------------------------------------------------------------
// RANDOMLY PASS OVER ALL SERVERS
```

```
Hashtable UDC = UDCnet.getNetwork();
int intSERVER = 0;
int intCOUNT = 0;
int[] intSERVERS = new int[UDC.size()];
boolean blnPassedOver = true;
for ( int intServer = 1; intServer <= UDC.size(); intServer++ )
{
intSERVERS[intServer-1] = -1;
}
int intTotalTasks = 0;
for ( int intServer = 1; intServer <= UDC.size(); intServer++ )
{
while ( blnPassedOver == true )
{
intSERVER = (int) (( random.nextDouble() * UDC.size() ) + 1 );
for ( int intArrServ = 1; intArrServ <= UDC.size();
intArrServ++ )
{
if ( intSERVERS[intArrServ-1] != intSERVER )
{
blnPassedOver = false;
}
}
}
blnPassedOver = true;
// add to array (to make sure it is not picked again)
intSERVERS[intCOUNT] = intSERVER;
UDC_Server server = (UDC_Server) UDC.get( "S" + intServer );
// RETRIEVE DATA FOR MBC VISUALISATIONS
dblVISload[intServer-1][intTimeStep-1] =
Double.parseDouble( "" + server.intSERVtotal_ru ) -
Double.parseDouble( "" + server.intSERVavail_ru );
Vector vctSERVmarket = server.vctSERVmarket;
dblVISno_tasks[intServer-1][intTimeStep-1] =
Double.parseDouble( "" + vctSERVmarket.size() );
// get visualisation stats from each server
dblVISpriceSS[intServer-1][intTimeStep-1] =
server.agtSERVseller.dblAGNTprice;
dblVISpriceSB[intServer-1][intTimeStep-1] =
server.agtSERVbuyer.dblAGNTprice;
dblVISjprsd[intServer-1][intTimeStep-1] = (double)
server.intSERVjprsd;
dblVISalloc[intServer-1][intTimeStep-1] = (double)
server.intSERValloc;
if ( dblVISno_tasks[intServer-1][intTimeStep-1] != 0 )
{
dblVISperf[intServer-1][intTimeStep-1] =
(double) ( server.intSERValloc /
( dblVISno_tasks[intServer-1][intTimeStep-1]
+ server.intSERValloc )) * 100;
}
else
{
dblVISperf[intServer-1][intTimeStep-1] = 0.0;
}
// update counter of total number of server tasks processed
if ( intTimeStep == ( intVISno_timesteps - 1 ) )
{
intTotalJobsPrsd += server.intSERVjprsd;
}
// update counter of total number of server tasks allocated
if ( intTimeStep == ( intVISno_timesteps - 1 ) )
{
intTotalJobsAlloc += server.intSERValloc;
}
if ( intTimeStep == ( intVISno_timesteps - 1 ) )
{
intTotalTasks += ( dblVISno_tasks[intServer-1][intTimeStep-1]
+ server.intSERValloc );
}
```

```
// ------------------------------------------------------------------
// act on local state of server only
// call thread to update the server state
intTASKid = server.updateServer( intTimeStep, intTASKid, blnOutput,
stats, random, psOUT, psSDO, dblAGNT_lr_L, dblAGNT_lr_H,
dblAGNT_mm_L, dblAGNT_mm_H, dblAGNT_pf_L, dblAGNT_pf_H,
dblAGNT_tr_R, dblAGNT_tr_A, dblPrice_0,dblMaxSurplus, UDCnet,
SDVIS, intVISno_timesteps );
// get from server whether a deal (trade) occured
if ( server.blnTradeOccured )
{
intTrades[intTimeStep-1]++;
dblSuccAucs += 1.0;
blnTrade[intTimeStep-1] = true;
}
// get from server whether an auction occured (successful or not)
if ( server.blnAuctionOccured )
{
dblAuctions += 1.0;
}
}
DecimalFormat decfor = new DecimalFormat( "0.00" );
if ( intTimeStep == ( intVISno_timesteps - 1 ) )
{
dblTaskAllocPerc = (double) ( (double) intTotalJobsAlloc
/ (double) intTotalTasks ) * 100.0;
dblTaskAllocPerc = Double.parseDouble( decfor.format(
dblTaskAllocPerc ) );
}
// update stats based on whether a deal occured
stats.STATS_update( intTimeStep, UDCnet, UDC.size(), UDC.size(),
blnTrade[intTimeStep-1], blnOutput );
// output visualisation
if ( intTimeStep == intVISno_timesteps )
{
UDC_Vis loadVIS = new UDC_Vis( intVISno_servers, intVISno_timesteps,
"LOAD", dblVISload, intTotalJobsPrsd, dblTaskAllocPerc );
UDC_Vis taskVIS = new UDC_Vis( intVISno_servers, intVISno_timesteps,
"WAITING TASKS", dblVISno_tasks, intTotalJobsPrsd,
dblTaskAllocPerc );
UDC_Vis SSpriceVIS = new UDC_Vis( intVISno_servers, intVISno_timesteps,
"OFFER PRICE", dblVISpriceSS, intTotalJobsPrsd, dblTaskAllocPerc );
UDC_Vis SBpriceVIS = new UDC_Vis( intVISno_servers, intVISno_timesteps,
"BID PRICE", dblVISpriceSB, intTotalJobsPrsd, dblTaskAllocPerc );
UDC_Vis jprsdVIS = new UDC_Vis( intVISno_servers, intVISno_timesteps,
"JOBS PRSD", dblVISjprsd, intTotalJobsPrsd, dblTaskAllocPerc );
UDC_Vis allocVIS = new UDC_Vis( intVISno_servers, intVISno_timesteps,
"TASKS ALLOCATED", dblVISalloc, intTotalJobsAlloc,
dblTaskAllocPerc );
UDC_Vis perfVIS = new UDC_Vis( intVISno_servers, intVISno_timesteps,
"PERCENTAGE ALLOCATED", dblVISperf, intTotalJobsPrsd,
dblTaskAllocPerc );
if ( blnSERVvis == true )
{
//taskVIS.show();
//jprsdVIS.show();
//allocVIS.show();
//perfVIS.show();
SBpriceVIS.show();
SSpriceVIS.show();
loadVIS.show();
}
stats.TRADE_output( intTimeStep, intTrades, psTRA, intGen );
stats.DAY_output( intVISno_timesteps, psDAY, blnTrade, intGen );
}
return intTASKid;
}
// ------------------------------------------------------------------
// SIM_BUILD_UDC: set-up the UDC_Network and populate with UDC_Server
```

```
// objects
public static UDC_Network SIM_build_udc( Random random,
boolean blnOutput )
{
UDC_Network UDCnet = new UDC_Network();
// INSTANTIATE UDC_SERVER OBJECTS
UDC_Server S1 = new UDC_Server( "S1", 1 );
UDC_Server S2 = new UDC_Server( "S2", 1 );
UDC_Server S3 = new UDC_Server( "S3", 1 );
UDC_Server S4 = new UDC_Server( "S4", 1 );
UDC_Server S5 = new UDC_Server( "S5", 1 );
UDC_Server S6 = new UDC_Server( "S6", 1 );
UDC_Server S7 = new UDC_Server( "S7", 1 );
UDC_Server S8 = new UDC_Server( "S8", 1 );
UDC_Server S9 = new UDC_Server( "S9", 1 );
UDC_Server S10 = new UDC_Server( "S10", 1 );
UDC_Server S11 = new UDC_Server( "S11", 1 );
UDC_Server S12 = new UDC_Server( "S12", 1 );
UDC_Server S13 = new UDC_Server( "S13", 1 );
UDC_Server S14 = new UDC_Server( "S14", 1 );
UDC_Server S15 = new UDC_Server( "S15", 1 );
UDC_Server S16 = new UDC_Server( "S16", 1 );
UDC_Server S17 = new UDC_Server( "S17", 1 );
UDC_Server S18 = new UDC_Server( "S18", 1 );
UDC_Server S19 = new UDC_Server( "S19", 1 );
UDC_Server S20 = new UDC_Server( "S20", 1 );
UDC_Server S21 = new UDC_Server( "S21", 1 );
UDC_Server S22 = new UDC_Server( "S22", 1 );
UDC_Server S23 = new UDC_Server( "S23", 1 );
UDC_Server S24 = new UDC_Server( "S24", 1 );
UDC_Server S25 = new UDC_Server( "S25", 1 );
UDC_Server S26 = new UDC_Server( "S26", 1 );
UDC_Server S27 = new UDC_Server( "S27", 1 );
UDC_Server S28 = new UDC_Server( "S28", 1 );
UDC_Server S29 = new UDC_Server( "S29", 1 );
UDC_Server S30 = new UDC_Server( "S30", 1 );
UDC_Server S31 = new UDC_Server( "S31", 1 );
UDC_Server S32 = new UDC_Server( "S32", 1 );
UDC_Server S33 = new UDC_Server( "S33", 1 );
UDC_Server S34 = new UDC_Server( "S34", 1 );
UDC_Server S35 = new UDC_Server( "S35", 1 );
UDC_Server S36 = new UDC_Server( "S36", 1 );
UDC_Server S37 = new UDC_Server( "S37", 1 );
UDC_Server S38 = new UDC_Server( "S38", 1 );
UDC_Server S39 = new UDC_Server( "S39", 1 );
UDC_Server S40 = new UDC_Server( "S40", 1 );
UDC_Server S41 = new UDC_Server( "S41", 1 );
UDC_Server S42 = new UDC_Server( "S42", 1 );
UDC_Server S43 = new UDC_Server( "S43", 1 );
UDC_Server S44 = new UDC_Server( "S44", 1 );
UDC_Server S45 = new UDC_Server( "S45", 1 );
UDC_Server S46 = new UDC_Server( "S46", 1 );
UDC_Server S47 = new UDC_Server( "S47", 1 );
UDC_Server S48 = new UDC_Server( "S48", 1 );
UDC_Server S49 = new UDC_Server( "S49", 1 );
UDC_Server S50 = new UDC_Server( "S50", 1 );
UDC_Server S51 = new UDC_Server( "S51", 1 );
UDC_Server S52 = new UDC_Server( "S52", 1 );
UDC_Server S53 = new UDC_Server( "S53", 1 );
UDC_Server S54 = new UDC_Server( "S54", 1 );
UDC_Server S55 = new UDC_Server( "S55", 1 );
UDC_Server S56 = new UDC_Server( "S56", 1 );
UDC_Server S57 = new UDC_Server( "S57", 1 );
UDC_Server S58 = new UDC_Server( "S58", 1 );
UDC_Server S59 = new UDC_Server( "S59", 1 );
UDC_Server S60 = new UDC_Server( "S60", 1 );
UDC_Server S61 = new UDC_Server( "S61", 1 );
UDC_Server S62 = new UDC_Server( "S62", 1 );
UDC_Server S63 = new UDC_Server( "S63", 1 );
UDC_Server S64 = new UDC_Server( "S64", 1 );
UDC_Server S65 = new UDC_Server( "S65", 1 );
```

```
UDC_Server S66 = new UDC_Server( "S66", 1 );
UDC_Server S67 = new UDC_Server( "S67", 1 );
UDC_Server S68 = new UDC_Server( "S68", 1 );
UDC_Server S69 = new UDC_Server( "S69", 1 );
UDC_Server S70 = new UDC_Server( "S70", 1 );
UDC_Server S71 = new UDC_Server( "S71", 1 );
UDC_Server S72 = new UDC_Server( "S72", 1 );
UDC_Server S73 = new UDC_Server( "S73", 1 );
UDC_Server S74 = new UDC_Server( "S74", 1 );
UDC_Server S75 = new UDC_Server( "S75", 1 );
UDC_Server S76 = new UDC_Server( "S76", 1 );
UDC_Server S77 = new UDC_Server( "S77", 1 );
UDC_Server S78 = new UDC_Server( "S78", 1 );
UDC_Server S79 = new UDC_Server( "S79", 1 );
UDC_Server S80 = new UDC_Server( "S80", 1 );
UDC_Server S81 = new UDC_Server( "S81", 1 );
UDC_Server S82 = new UDC_Server( "S82", 1 );
UDC_Server S83 = new UDC_Server( "S83", 1 );
UDC_Server S84 = new UDC_Server( "S84", 1 );
UDC_Server S85 = new UDC_Server( "S85", 1 );
UDC_Server S86 = new UDC_Server( "S86", 1 );
UDC_Server S87 = new UDC_Server( "S87", 1 );
UDC_Server S88 = new UDC_Server( "S88", 1 );
UDC_Server S89 = new UDC_Server( "S89", 1 );
UDC_Server S90 = new UDC_Server( "S90", 1 );
UDC_Server S91 = new UDC_Server( "S91", 1 );
UDC_Server S92 = new UDC_Server( "S92", 1 );
UDC_Server S93 = new UDC_Server( "S93", 1 );
UDC_Server S94 = new UDC_Server( "S94", 1 );
UDC_Server S95 = new UDC_Server( "S95", 1 );
UDC_Server S96 = new UDC_Server( "S96", 1 );
UDC_Server S97 = new UDC_Server( "S97", 1 );
UDC_Server S98 = new UDC_Server( "S98", 1 );
UDC_Server S99 = new UDC_Server( "S99", 1 );
UDC_Server S100 = new UDC_Server( "S100", 1 );
/*
// randomly determine the total resource units available on each
// server
// (range [2,10]) to initialise each server (9 is exclusive below)
UDC_Server S1 = new UDC_Server( "S1",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S2 = new UDC_Server( "S2",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S3 = new UDC_Server( "S3",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S4 = new UDC_Server( "S4",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S5 = new UDC_Server( "S5",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S6 = new UDC_Server( "S6",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S7 = new UDC_Server( "S7",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S8 = new UDC_Server( "S8",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S9 = new UDC_Server( "S9",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S10 = new UDC_Server( "S10",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S11 = new UDC_Server( "S11",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S12 = new UDC_Server( "S12",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S13 = new UDC_Server( "S13",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S14 = new UDC_Server( "S14",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S15 = new UDC_Server( "S15",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S16 = new UDC_Server( "S16",
( random.nextInt( 9 ) + 2 ) );
```

```
UDC_Server S17 = new UDC_Server( "S17",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S18 = new UDC_Server( "S18",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S19 = new UDC_Server( "S19",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S20 = new UDC_Server( "S20",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S21 = new UDC_Server( "S21",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S22 = new UDC_Server( "S22",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S23 = new UDC_Server( "S23",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S24 = new UDC_Server( "S24",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S25 = new UDC_Server( "S25",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S26 = new UDC_Server( "S26",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S27 = new UDC_Server( "S27",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S28 = new UDC_Server( "S28",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S29 = new UDC_Server( "S29",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S30 = new UDC_Server( "S30",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S31 = new UDC_Server( "S31",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S32 = new UDC_Server( "S32",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S33 = new UDC_Server( "S33",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S34 = new UDC_Server( "S34",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S35 = new UDC_Server( "S35",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S36 = new UDC_Server( "S36",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S37 = new UDC_Server( "S37",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S38 = new UDC_Server( "S38",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S39 = new UDC_Server( "S39",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S40 = new UDC_Server( "S40",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S41 = new UDC_Server( "S41",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S42 = new UDC_Server( "S42",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S43 = new UDC_Server( "S43",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S44 = new UDC_Server( "S44",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S45 = new UDC_Server( "S45",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S46 = new UDC_Server( "S46",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S47 = new UDC_Server( "S47",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S48 = new UDC_Server( "S48",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S49 = new UDC_Server( "S49",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S50 = new UDC_Server( "S50",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S51 = new UDC_Server( "S51",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S52 = new UDC_Server( "S52",
```

```
( random.nextInt( 9 ) + 2 ) );
UDC_Server S53 = new UDC_Server( "S53",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S54 = new UDC_Server( "S54",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S55 = new UDC_Server( "S55",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S56 = new UDC_Server( "S56",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S57 = new UDC_Server( "S57",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S58 = new UDC_Server( "S58",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S59 = new UDC_Server( "S59",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S60 = new UDC_Server( "S60",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S61 = new UDC_Server( "S61",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S62 = new UDC_Server( "S62",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S63 = new UDC_Server( "S63",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S64 = new UDC_Server( "S64",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S65 = new UDC_Server( "S65",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S66 = new UDC_Server( "S66",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S67 = new UDC_Server( "S67",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S68 = new UDC_Server( "S68",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S69 = new UDC_Server( "S69",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S70 = new UDC_Server( "S70",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S71 = new UDC_Server( "S71",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S72 = new UDC_Server( "S72",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S73 = new UDC_Server( "S73",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S74 = new UDC_Server( "S74",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S75 = new UDC_Server( "S75",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S76 = new UDC_Server( "S76",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S77 = new UDC_Server( "S77",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S78 = new UDC_Server( "S78",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S79 = new UDC_Server( "S79",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S80 = new UDC_Server( "S80",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S81 = new UDC_Server( "S81",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S82 = new UDC_Server( "S82",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S83 = new UDC_Server( "S83",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S84 = new UDC_Server( "S84",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S85 = new UDC_Server( "S85",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S86 = new UDC_Server( "S86",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S87 = new UDC_Server( "S87",
( random.nextInt( 9 ) + 2 ) );
```

```
UDC_Server S88 = new UDC_Server( "S88",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S89 = new UDC_Server( "S89",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S90 = new UDC_Server( "S90",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S91 = new UDC_Server( "S91",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S92 = new UDC_Server( "S92",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S93 = new UDC_Server( "S93",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S94 = new UDC_Server( "S94",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S95 = new UDC_Server( "S95",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S96 = new UDC_Server( "S96",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S97 = new UDC_Server( "S97",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S98 = new UDC_Server( "S98",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S99 = new UDC_Server( "S99",
( random.nextInt( 9 ) + 2 ) );
UDC_Server S100 = new UDC_Server( "S100",
( random.nextInt( 9 ) + 2 ) );
*/
// ADD SERVERS TO UDC_NETWORK
UDCnet.put( S1 );
UDCnet.put( S2 );
UDCnet.put( S3 );
UDCnet.put( S4 );
UDCnet.put( S5 );
UDCnet.put( S6 );
UDCnet.put( S7 );
UDCnet.put( S8 );
UDCnet.put( S9 );
UDCnet.put( S10 );
UDCnet.put( S11 );
UDCnet.put( S12 );
UDCnet.put( S13 );
UDCnet.put( S14 );
UDCnet.put( S15 );
UDCnet.put( S16 );
UDCnet.put( S17 );
UDCnet.put( S18 );
UDCnet.put( S19 );
UDCnet.put( S20 );
UDCnet.put( S21 );
UDCnet.put( S22 );
UDCnet.put( S23 );
UDCnet.put( S24 );
UDCnet.put( S25 );
UDCnet.put( S26 );
UDCnet.put( S27 );
UDCnet.put( S28 );
UDCnet.put( S29 );
UDCnet.put( S30 );
UDCnet.put( S31 );
UDCnet.put( S32 );
UDCnet.put( S33 );
UDCnet.put( S34 );
UDCnet.put( S35 );
UDCnet.put( S36 );
UDCnet.put( S37 );
UDCnet.put( S38 );
UDCnet.put( S39 );
```

```
UDCnet.put( S40 );
UDCnet.put( S41 );
UDCnet.put( S42 );
UDCnet.put( S43 );
UDCnet.put( S44 );
UDCnet.put( S45 );
UDCnet.put( S46 );
UDCnet.put( S47 );
UDCnet.put( S48 );
UDCnet.put( S49 );
UDCnet.put( S50 );
UDCnet.put( S51 );
UDCnet.put( S52 );
UDCnet.put( S53 );
UDCnet.put( S54 );
UDCnet.put( S55 );
UDCnet.put( S56 );
UDCnet.put( S57 );
UDCnet.put( S58 );
UDCnet.put( S59 );
UDCnet.put( S60 );
UDCnet.put( S61 );
UDCnet.put( S62 );
UDCnet.put( S63 );
UDCnet.put( S64 );
UDCnet.put( S65 );
UDCnet.put( S66 );
UDCnet.put( S67 );
UDCnet.put( S68 );
UDCnet.put( S69 );
UDCnet.put( S70 );
UDCnet.put( S71 );
UDCnet.put( S72 );
UDCnet.put( S73 );
UDCnet.put( S74 );
UDCnet.put( S75 );
UDCnet.put( S76 );
UDCnet.put( S77 );
UDCnet.put( S78 );
UDCnet.put( S79 );
UDCnet.put( S80 );
UDCnet.put( S81 );
UDCnet.put( S82 );
UDCnet.put( S83 );
UDCnet.put( S84 );
UDCnet.put( S85 );
UDCnet.put( S86 );
UDCnet.put( S87 );
UDCnet.put( S88 );
UDCnet.put( S89 );
UDCnet.put( S90 );
UDCnet.put( S91 );
UDCnet.put( S92 );
UDCnet.put( S93 );
UDCnet.put( S94 );
UDCnet.put( S95 );
UDCnet.put( S96 );
UDCnet.put( S97 );
UDCnet.put( S98 );
UDCnet.put( S99 );
UDCnet.put( S100 );
Hashtable UDC = UDCnet.getNetwork();
Enumeration enum = UDC.elements();
if ( blnOutput )
{
```

```
while ( enum.hasMoreElements() )
{
UDC_Server server = (UDC_Server) enum.nextElement();
//psOUT.println( "MBC_SIM: Server " + server.strSERVid
// + " with capacity " + server.intSERVtotal_ru +
// " initialised" );
}
}
// ADD SERVER CONNECTIVITIES
// (20-SERVER CONNECTIVITIES)
/*
S1.addUDC_Servers( S16, S17, S18, S19, S20,  S2,  S3,  S4,
S5,  S6 );
S2.addUDC_Servers( S17, S18, S19, S20,  S1,  S3,  S4,  S5,
S6,  S7 );
S3.addUDC_Servers( S18, S19, S20,  S1,  S2,  S4,  S5,  S6,
S7,  S8 );
S4.addUDC_Servers( S19, S20,  S1,  S2,  S3,  S5,  S6,  S7,
S8,  S9 );
S5.addUDC_Servers( S20,  S1,  S2,  S3,  S4,  S6,  S7,  S8,
S9, S10 );
S6.addUDC_Servers(  S1,  S2,  S3,  S4,  S5,  S7,  S8,  S9,
S10, S11 );
S7.addUDC_Servers(  S2,  S3,  S4,  S5,  S6,  S8,  S9, S10,
S11, S12 );
S8.addUDC_Servers(  S3,  S4,  S5,  S6,  S7,  S9, S10, S11,
S12, S13 );
S9.addUDC_Servers(  S4,  S5,  S6,  S7,  S8, S10, S11, S12,
S13, S14 );
S10.addUDC_Servers(  S5,  S6,  S7,  S8,  S9, S11, S12, S13,
S14, S15 );
S11.addUDC_Servers(  S6,  S7,  S8,  S9, S10, S12, S13, S14,
S15, S16 );
S12.addUDC_Servers(  S7,  S8,  S9, S10, S11, S13, S14, S15,
S16, S17 );
S13.addUDC_Servers(  S8,  S9, S10, S11, S12, S14, S15, S16,
S17, S18 );
S14.addUDC_Servers(  S9, S10, S11, S12, S13, S15, S16, S17,
S18, S19 );
S15.addUDC_Servers( S10, S11, S12, S13, S14, S16, S17, S18,
S19, S20 );
S16.addUDC_Servers( S11, S12, S13, S14, S15, S17, S18, S19,
S20, S1 );
S17.addUDC_Servers( S12, S13, S14, S15, S16, S18, S19, S20,
S1, S2 );
S18.addUDC_Servers( S13, S14, S15, S16, S17, S19, S20, S1,
S2, S3 );
S19.addUDC_Servers( S14, S15, S16, S17, S18, S20, S1, S2,
S3, S4 );
S20.addUDC_Servers( S15, S16, S17, S18, S19, S1, S2, S3,
S4, S5 );
*/
// (100-SERVER CONNECTIVITIES)
S1.addUDC_Servers( S96, S97, S98, S99, S100,  S2,  S3,  S4,
S5,  S6 );
S2.addUDC_Servers( S97, S98, S99, S100,  S1,  S3,  S4,  S5,
S6,  S7 );
S3.addUDC_Servers( S98, S99, S100,  S1,  S2,  S4,  S5,  S6,
S7,  S8 );
S4.addUDC_Servers( S99, S100,  S1,  S2,  S3,  S5,  S6,  S7,
S8,  S9 );
S5.addUDC_Servers( S100,  S1,  S2,  S3,  S4,  S6,  S7,  S8,
S9, S10 );
S6.addUDC_Servers(  S1,  S2,  S3,  S4,  S5,  S7,  S8,  S9,
S10, S11 );
S7.addUDC_Servers(  S2,  S3,  S4,  S5,  S6,  S8,  S9, S10,
S11, S12 );
S8.addUDC_Servers(  S3,  S4,  S5,  S6,  S7,  S9, S10, S11,
S12, S13 );
S9.addUDC_Servers(  S4,  S5,  S6,  S7,  S8, S10, S11, S12,
S13, S14 );
```

```
S10.addUDC_Servers(  S5,  S6,  S7,  S8,  S9, S11, S12, S13,
S14, S15 );
S11.addUDC_Servers(  S6,  S7,  S8,  S9, S10, S12, S13, S14,
S15, S16 );
S12.addUDC_Servers(  S7,  S8,  S9, S10, S11, S13, S14, S15,
S16, S17 );
S13.addUDC_Servers(  S8,  S9, S10, S11, S12, S14, S15, S16,
S17, S18 );
S14.addUDC_Servers(  S9, S10, S11, S12, S13, S15, S16, S17,
S18, S19 );
S15.addUDC_Servers( S10, S11, S12, S13, S14, S16, S17, S18,
S19, S20 );
S16.addUDC_Servers( S11, S12, S13, S14, S15, S17, S18, S19,
S20, S21 );
S17.addUDC_Servers( S12, S13, S14, S15, S16, S18, S19, S20,
S21, S22 );
S18.addUDC_Servers( S13, S14, S15, S16, S17, S19, S20, S21,
S22, S23 );
S19.addUDC_Servers( S14, S15, S16, S17, S18, S20, S21, S22,
S23, S24 );
S20.addUDC_Servers( S15, S16, S17, S18, S19, S21, S22, S23,
S24, S25 );
S21.addUDC_Servers( S16, S17, S18, S19, S20, S22, S23, S24,
S25, S26 );
S22.addUDC_Servers( S17, S18, S19, S20, S21, S23, S24, S25,
S26, S27 );
S23.addUDC_Servers( S18, S19, S20, S21, S22, S24, S25, S26,
S27, S28 );
S24.addUDC_Servers( S19, S20, S21, S22, S23, S25, S26, S27,
S28, S29 );
S25.addUDC_Servers( S20, S21, S22, S23, S24, S26, S27, S28,
S29, S30 );
S26.addUDC_Servers( S21, S22, S23, S24, S25, S27, S28, S29,
S30, S31 );
S27.addUDC_Servers( S22, S23, S24, S25, S26, S28, S29, S30,
S31, S32 );
S28.addUDC_Servers( S23, S24, S25, S26, S27, S29, S30, S31,
S32, S33 );
S29.addUDC_Servers( S24, S25, S26, S27, S28, S30, S31, S32,
S33, S34 );
S30.addUDC_Servers( S25, S26, S27, S28, S29, S31, S32, S33,
S34, S35 );
S31.addUDC_Servers( S26, S27, S28, S29, S30, S32, S33, S34,
S35, S36 );
S32.addUDC_Servers( S27, S28, S29, S30, S31, S33, S34, S35,
S36, S37 );
S33.addUDC_Servers( S28, S29, S30, S31, S32, S34, S35, S36,
S37, S38 );
S34.addUDC_Servers( S29, S30, S31, S32, S33, S35, S36, S37,
S38, S39 );
S35.addUDC_Servers( S30, S31, S32, S33, S34, S36, S37, S38,
S39, S40 );
S36.addUDC_Servers( S31, S32, S33, S34, S35, S37, S38, S39,
S40, S41 );
S37.addUDC_Servers( S32, S33, S34, S35, S36, S38, S39, S40,
S41, S42 );
S38.addUDC_Servers( S33, S34, S35, S36, S37, S39, S40, S41,
S42, S43 );
S39.addUDC_Servers( S34, S35, S36, S37, S38, S40, S41, S42,
S43, S44 );
S40.addUDC_Servers( S35, S36, S37, S38, S39, S41, S42, S43,
S44, S45 );
S41.addUDC_Servers( S36, S37, S38, S39, S40, S42, S43, S44,
S45, S46 );
S42.addUDC_Servers( S37, S38, S39, S40, S41, S43, S44, S45,
S46, S47 );
S43.addUDC_Servers( S38, S39, S40, S41, S42, S44, S45, S46,
S47, S48 );
S44.addUDC_Servers( S39, S40, S41, S42, S43, S45, S46, S47,
S48, S49 );
S45.addUDC_Servers( S40, S41, S42, S43, S44, S46, S47, S48,
```

```
S49, S50 );
S46.addUDC_Servers( S41, S42, S43, S44, S45, S47, S48, S49,
S50, S51 );
S47.addUDC_Servers( S42, S43, S44, S45, S46, S48, S49, S50,
S51, S52 );
S48.addUDC_Servers( S43, S44, S45, S46, S47, S49, S50, S51,
S52, S53 );
S49.addUDC_Servers( S44, S45, S46, S47, S48, S50, S51, S52,
S53, S54 );
S50.addUDC_Servers( S45, S46, S47, S48, S49, S51, S52, S53,
S54, S55 );
S51.addUDC_Servers( S46, S47, S48, S49, S50, S52, S53, S54,
S55, S56 );
S52.addUDC_Servers( S47, S48, S49, S50, S51, S53, S54, S55,
S56, S57 );
S53.addUDC_Servers( S48, S49, S50, S51, S52, S54, S55, S56,
S57, S58 );
S54.addUDC_Servers( S49, S50, S51, S52, S53, S55, S56, S57,
S58, S59 );
S55.addUDC_Servers( S50, S51, S52, S53, S54, S56, S57, S58,
S59, S60 );
S56.addUDC_Servers( S51, S52, S53, S54, S55, S57, S58, S59,
S60, S61 );
S57.addUDC_Servers( S52, S53, S54, S55, S56, S58, S59, S60,
S61, S62 );
S58.addUDC_Servers( S53, S54, S55, S56, S57, S59, S60, S61,
S62, S63 );
S59.addUDC_Servers( S54, S55, S56, S57, S58, S60, S61, S62,
S63, S64 );
S60.addUDC_Servers( S55, S56, S57, S58, S59, S61, S62, S63,
S64, S65 );
S61.addUDC_Servers( S56, S57, S58, S59, S60, S62, S63, S64,
S65, S66 );
S62.addUDC_Servers( S57, S58, S59, S60, S61, S63, S64, S65,
S66, S67 );
S63.addUDC_Servers( S58, S59, S60, S61, S62, S64, S65, S66,
S67, S68 );
S64.addUDC_Servers( S59, S60, S61, S62, S63, S65, S66, S67,
S68, S69 );
S65.addUDC_Servers( S60, S61, S62, S63, S64, S66, S67, S68,
S69, S70 );
S66.addUDC_Servers( S61, S62, S63, S64, S65, S67, S68, S69,
S70, S71 );
S67.addUDC_Servers( S62, S63, S64, S65, S66, S68, S69, S70,
S71, S72 );
S68.addUDC_Servers( S63, S64, S65, S66, S67, S69, S70, S71,
S72, S73 );
S69.addUDC_Servers( S64, S65, S66, S67, S68, S70, S71, S72,
S73, S74 );
S70.addUDC_Servers( S65, S66, S67, S68, S69, S71, S72, S73,
S74, S75 );
S71.addUDC_Servers( S66, S67, S68, S69, S70, S72, S73, S74,
S75, S76 );
S72.addUDC_Servers( S67, S68, S69, S70, S71, S73, S74, S75,
S76, S77 );
S73.addUDC_Servers( S68, S69, S70, S71, S72, S74, S75, S76,
S77, S78 );
S74.addUDC_Servers( S69, S70, S71, S72, S73, S75, S76, S77,
S78, S79 );
S75.addUDC_Servers( S70, S71, S72, S73, S74, S76, S77, S78,
S79, S80 );
S76.addUDC_Servers( S71, S72, S73, S74, S75, S77, S78, S79,
S80, S81 );
S77.addUDC_Servers( S72, S73, S74, S75, S76, S78, S79, S80,
S81, S82 );
S78.addUDC_Servers( S73, S74, S75, S76, S77, S79, S80, S81,
S82, S83 );
S79.addUDC_Servers( S74, S75, S76, S77, S78, S80, S81, S82,
S83, S84 );
S80.addUDC_Servers( S75, S76, S77, S78, S79, S81, S82, S83,
S84, S85 );
```

```
S81.addUDC_Servers( S76, S77, S78, S79, S80, S82, S83, S84,
S85, S86 );
S82.addUDC_Servers( S77, S78, S79, S80, S81, S83, S84, S85,
S86, S87 );
S83.addUDC_Servers( S78, S79, S80, S81, S82, S84, S85, S86,
S87, S88 );
S84.addUDC_Servers( S79, S80, S81, S82, S83, S85, S86, S87,
S88, S89 );
S85.addUDC_Servers( S80, S81, S82, S83, S84, S86, S87, S88,
S89, S90 );
S86.addUDC_Servers( S81, S82, S83, S84, S85, S87, S88, S89,
S90, S91 );
S87.addUDC_Servers( S82, S83, S84, S85, S86, S88, S89, S90,
S91, S92 );
S88.addUDC_Servers( S83, S84, S85, S86, S87, S89, S90, S91,
S92, S93 );
S89.addUDC_Servers( S84, S85, S86, S87, S88, S90, S91, S92,
S93, S94 );
S90.addUDC_Servers( S85, S86, S87, S88, S89, S91, S92, S93,
S94, S95 );
S91.addUDC_Servers( S86, S87, S88, S89, S90, S92, S93, S94,
S95, S96 );
S92.addUDC_Servers( S87, S88, S89, S90, S91, S93, S94, S95,
S96, S97 );
S93.addUDC_Servers( S88, S89, S90, S91, S92, S94, S95, S96,
S97, S98 );
S94.addUDC_Servers( S89, S90, S91, S92, S93, S95, S96, S97,
S98, S99 );
S95.addUDC_Servers( S90, S91, S92, S93, S94, S96, S97, S98,
S99, S100 );
S96.addUDC_Servers( S91, S92, S93, S94, S95, S97, S98, S99,
S100,  S1 );
S97.addUDC_Servers( S92, S93, S94, S95, S96, S98, S99, S100,
S1,  S2 );
S98.addUDC_Servers( S93, S94, S95, S96, S97, S99, S100,  S1,
S2,  S3 );
S99.addUDC_Servers( S94, S95, S96, S97, S98, S100,  S1,  S2,
S3,  S4 );
S100.addUDC_Servers( S95, S96, S97, S98, S99, S1,  S2,  S3,
S4,  S5 );
return UDCnet;
}
// ---------------------------------------------------------------------
// RESET_UDC: reset UDC_Server parameters for the next trial
public void reset_UDC( UDC_Network UDCnet, int intVISno_timesteps )
{
// reset the UDC network parameters -- i.e. server state; tasks
// running on the server; all visualisation parameters such as
// load etc.
Hashtable UDC = UDCnet.getNetwork();
for ( int intServer = 1; intServer <= UDC.size(); intServer++ )
{
UDC_Server server = (UDC_Server) UDC.get( "S" + intServer );
server.intSERVstate = 0;
server.vctSERVmarket.clear();
server.vctSERVtasks.clear();
server.intSERVavail_ru = server.intSERVtotal_ru;
server.blnSERVret_willing = false;
server.blnSERVwilling = false;
server.blnWillingWaitType = false;
server.intNoStepsWaiting1 = 0;
server.intNoStepsWaiting2 = 0;
server.blnStopAuction = false;
server.blnAuctionOccured = false;
server.blnTradeOccured = false;
server.intSERVid1 = 0;
//server.blnEnableShifting = false;
}
dblAuctions = 0.0;
dblSuccAucs = 0.0;
intTimeStep = 0;
```

```
intTotalJobsPrsd = 0;
intTotalJobsAlloc = 0;
dblTaskAllocPerc = 0.0;
dblVISload = new double[UDC.size()][intVISno_timesteps];
dblVISno_tasks = new double[UDC.size()][intVISno_timesteps];
dblVISpriceSS = new double[UDC.size()][intVISno_timesteps];
dblVISpriceSB = new double[UDC.size()][intVISno_timesteps];
dblVISjprsd = new double[UDC.size()][intVISno_timesteps];
dblVISalloc = new double[UDC.size()][intVISno_timesteps];
dblVISperf = new double[UDC.size()][intVISno_timesteps];
for ( int intServer = 0; intServer < intVISno_servers;
intServer++ )
{
for ( int intTimeStep = 0; intTimeStep < intVISno_timesteps;
intTimeStep++ )
{
dblVISload[intServer][intTimeStep] = 0.0;
dblVISno_tasks[intServer][intTimeStep] = 0.0;
dblVISpriceSS[intServer][intTimeStep] = 0.0;
dblVISpriceSB[intServer][intTimeStep] = 0.0;
dblVISjprsd[intServer][intTimeStep] = 0.0;
dblVISalloc[intServer][intTimeStep] = 0.0;
dblVISperf[intServer][intTimeStep] = 0.0;
}
}
}
// ---------------------------------------------------------------------
// SIM_INIT: set agent parameters
// >> called by MBC_Sim.SIM_main
private void SIM_init( boolean blnOutput, PrintStream psOUT,
ZIP_SD_Vis SDVIS, UDC_Network UDCnet, Random random,
PrintStream psSDO )
{
Hashtable UDC = UDCnet.getNetwork();
// ---------------------------------------------------------------------
// SET NUMBER OF BUYER AND SELLER AGENTS
// (one for each UDC_Server object)
int intNoBuyers = UDC.size();
int intNoSellers = UDC.size();
// iterate through the buyer and seller agents attached to the servers
for ( int intServer = 1; intServer <= UDC.size(); intServer++ )
{
// retrieve network server object
UDC_Server server = (UDC_Server) UDC.get( "S" + intServer );
ZIP_Agent agtSERVbuyer = server.agtSERVbuyer;
ZIP_Agent agtSERVseller = server.agtSERVseller;
// ---------------------------------------------------------------------
// SET AGENT AS ACTIVE
agtSERVbuyer.dblAGNTactualgain = 0.0;
agtSERVseller.dblAGNTactualgain = 0.0;
// ---------------------------------------------------------------------
// SET AGENT LIMIT PRICES
// server buyer agents initially set a value within the range of the
// demand curve (until it needs to purchase some resources, then the
// limit price equals whatever price the server's seller got for this
// job that now wants to be unloaded)
agtSERVbuyer.dblAGNTlimit = ( random.nextDouble() * 2.5 ) + 0.75;
//( random.nextDouble() * 2.5 ) + 0.75;
// initially set server limit price in proportion to the *total* amount
// of resource units the server has -- i.e. re-scale the total resource
// units (range [2,10]) to [0.75, 3.25].
// (if all servers have the same total resource unit capacity
// this creates a flat supply curve)
agtSERVseller.dblAGNTlimit = server.intSERVtotal_ru * 0.325;
// server.intSERVtotal_ru * 0.325;
// ---------------------------------------------------------------------
// CALCULATE PRICE OF AGENT FROM LIMIT AND PROFIT VALUES
agtSERVbuyer.AGNT_calcPrice();
```

```
agtSERVseller.AGNT_calcPrice();
if ( blnOutput )
{
psOUT.println( "MBC_SIM.init_day: Buyer on Server "
+ server.strSERVid + ": price " + agtSERVbuyer.dblAGNTprice );
psOUT.println( "MBC_SIM.init_day: Seller on Server "
+ server.strSERVid + ": price " + agtSERVseller.dblAGNTprice );
}
}
// ---------------------------------------------------------------------
// IDENTIFY THEORETICAL EQUILIBRIUM PRICE
int intQ_0 = 0;
double dblEqProfit;
SDVIS.SDVIS_sup_dem( intNoSellers, intNoBuyers, UDCnet, dblPrice_0,
intQ_0, dblMaxSurplus, EQ_THEORY, 0.0, blnOutput, psSDO );
dblPrice_0 = SDVIS.dblEqPrice;
intQ_0 = SDVIS.intIQuant;
dblMaxSurplus = SDVIS.dblSurplus;
// ---------------------------------------------------------------------
// SET THEORETICAL GAINS FOR BUYERS AND SELLERS
for ( int intServer = 1; intServer <= UDC.size(); intServer++ )
{
// retrieve network server object
UDC_Server server = (UDC_Server) UDC.get( "S" + intServer );
ZIP_Agent agtSERVbuyer = new ZIP_Agent();
agtSERVbuyer = server.agtSERVbuyer;
dblEqProfit = agtSERVbuyer.dblAGNTlimit - dblPrice_0;
if ( dblEqProfit < 0.0 )
{
dblEqProfit = 0.0;
}
agtSERVbuyer.dblAGNTtheorgain = dblEqProfit;
}
for ( int intServer = 1; intServer <= UDC.size(); intServer++ )
{
// retrieve network server object
UDC_Server server = (UDC_Server) UDC.get( "S" + intServer );
ZIP_Agent agtSERVseller = new ZIP_Agent();
agtSERVseller = server.agtSERVseller;
dblEqProfit = agtSERVseller.dblAGNTlimit - dblPrice_0;
if ( dblEqProfit < 0.0 )
{
dblEqProfit = 0.0;
}
agtSERVseller.dblAGNTtheorgain = dblEqProfit;
}
}
// ---------------------------------------------------------------------
}
```

C.2.3 MBC_Stats.java

Description: Calculates and records trade and timestep statistics for the MBC simulation.

```
// Class:  MBC_Stats.java
import java.io.PrintStream;
import java.text.DecimalFormat;
import java.util.Hashtable;
public class MBC_Stats implements ZIP_Constants
{
// (initialised for use below)
private double dblPrice = 0.0;
// sum the s.d. of the transaction (deal) price around the
// equilibrium price
private double dblSigmaSum;
// smith's alpha (initialised to send to data_day_update method below)
private double dblAlpha = 0.0;
// equilibrium price
private double dblEqPrice;
// the last deal price
private double dblLastPrice = 0.0;
// sum of the price difference between the last deal price
// and the current deal price
private double dblPriceDiffSum;
// sum of the deal prices
private double dblPriceSum = 0;
// difference between actual gain and theoretical gain
private double dblDiff;
// square difference between actual gain and theoretical gain
private double dblPD;
// profit dispersal
private double dblProfitDisp;
// s.d. of the transaction price around the equilibrium price
// (used to calculate smith's alpha)
private double dblPDS;
// (initialised to send to data_day_update method below)
private double dblEfficiency = 0.0;
// trial fitness score
public double dblTrialFitness = 0.0;
// number of trades done in a timestep
private int intTradeNo;
// number of trades carried out over a trial
private int intNoTrades;
// equilibrium quantity
private int intEqQuantity;
// day (timestep) data array
private ZIP_Data_Day[] DayData;
// trade data array
private ZIP_Data_Trade[][] TradeData;
// trial stat arrays
private double[] TMPdblSum;
private double[] TMPdblSumSq;
private int[] TMPintN;
// ---------------------------------------------------------------------
// STATS_RESET: reset stat parameters
public void STATS_reset()
{
dblSigmaSum = 0.0;
dblPriceSum = 0.0;
dblPriceDiffSum = 0.0;
intTradeNo = 0;
}
// ---------------------------------------------------------------------
// STAT_INIT: initialise trade and timestep stat data structures
public void STATS_init( int intTimeSteps )
{
```

```
TMPdblSum = new double[intTimeSteps];
TMPdblSumSq = new double[intTimeSteps];
TMPintN = new int[intTimeSteps];
// initialise TradeData and DayData arrays
DayData = new ZIP_Data_Day[intTimeSteps];
TradeData = new ZIP_Data_Trade[intTimeSteps][1000];
for ( int intTimeStep = 0; intTimeStep < intTimeSteps;
intTimeStep++ )
{
DayData[intTimeStep] = new ZIP_Data_Day();
DayData[intTimeStep].data_day_init();
for ( int intServer = 0; intServer < 1000; intServer++ )
{
TradeData[intTimeStep][intServer] = new ZIP_Data_Trade();
// initially set deal prices to -1
TradeData[intTimeStep][intServer].dblTDATdeal_price = -1;
}
}
}
// ---------------------------------------------------------------------
// TRADE_UPDATE_THE_EQ: update theoretical equilibrium price
public void TRADE_update_the_eq( int intTimeStep, double dblEqPrice,
int intEqQuantity )
{
if ( dblEqPrice != NULL_EQ )
{
TradeData[intTimeStep-1][intTradeNo].dblTDATtheor_eq_price
= dblEqPrice;
TradeData[intTimeStep-1][intTradeNo].intTDATtheor_eq_quant
= intEqQuantity;
}
else
{
TradeData[intTimeStep-1][intTradeNo].intTDATtheor_eq_quant
= NULL_EQ;
}
}
// ---------------------------------------------------------------------
// TRADE_UPDATE_ACT_EQ: update actual equilibrium price
public void TRADE_updade_act_eq( int intTimeStep, double dblEqPrice,
int intEqQuantity )
{
if ( dblEqPrice != NULL_EQ )
{
TradeData[intTimeStep-1][intTradeNo].dblTDATactual_eq_price
= dblEqPrice;
TradeData[intTimeStep-1][intTradeNo].intTDATactual_eq_quant
= intEqQuantity;
}
else
{
TradeData[intTimeStep-1][intTradeNo].dblTDATactual_eq_price
= NULL_EQ;
}
}
// ---------------------------------------------------------------------
// TRADE_DEAL_TYPE_PRICE: record trade price and deal type
public void TRADE_deal_type_price( int intTimeStep, double dblPrice,
boolean blnDealType )
{
TradeData[intTimeStep-1][intTradeNo].dblTDATdeal_price = dblPrice;
TradeData[intTimeStep-1][intTradeNo].blnTDATdeal_type = blnDealType;
}
// ---------------------------------------------------------------------
// TRADE_NO_DEAL: no deal so set negative price in trade data
public void TRADE_no_deal( int intTimeStep )
{
TradeData[intTimeStep-1][intTradeNo].dblTDATdeal_price = -1.0;
}
```

```
// ---------------------------------------------------------------
// STATS_CALCULATE: if a buyer and a seller entered into a trade
// (a deal occured), calculate relevant stats
public void STATS_calculate( int intTimeStep, int intStatus,
double dblPrice, double dblPrice_0, double dblSurplus,
double dblMaxSurplus, boolean blnOutput )
{
if ( intStatus == DEAL )
{
// obtain the last deal price
if ( intNoTrades > 0 )
{
dblLastPrice =
TradeData[intTimeStep-1][intNoTrades-1].dblTDATdeal_price;
}
// sum the price difference between the last deal price and the
// current deal price (used to calculate volatility)
if ( intNoTrades > 0 )
{
dblPriceDiffSum += (( dblPrice - dblLastPrice ) * ( dblPrice -
dblLastPrice ));
}
// calculate s.d. of the transaction (deal) price around the
// equilibrium price (nb. dblPrice_0 is the *theoretical*
// eq. price)
dblPDS = (( dblPrice - dblPrice_0 ) * ( dblPrice - dblPrice_0 ));
// sum current deal prices
dblPriceSum += dblPrice;
// sum current s.d. of the transaction (deal) price around the
// equilibrium price
dblSigmaSum += dblPDS;
intTradeNo++;
intNoTrades++;
if ( intTradeNo != 0 )
{
// calculate alpha
dblAlpha = ( 100 * Math.sqrt( dblSigmaSum / intNoTrades ))
/ dblPrice_0;
}
// (to get round error on calculating square root)
else if ( intTradeNo == 0 )
{
// calculate alpha
dblAlpha = ( 100 * Math.sqrt( dblSigmaSum / 0.0000001 ))
/ dblPrice_0;
}
//  calculate efficiency
dblEfficiency = ( dblSurplus / dblMaxSurplus ) * 100;
if ( blnOutput )
{
//psOUT.println( "ZIP_SIM: DEAL " + intTradeNo + " alpha " +
// dblAlpha + " efficiency " + dblEfficiency );
}
}
else
{
// if no deal set alpha parameter to a max. value
dblAlpha = 100.0;
}
}
// ---------------------------------------------------------------
// STATS_UPDATE: update trading statistics
public void STATS_update( int intTimeStep, UDC_Network UDCnet,
int intNoBuyers, int intNoSellers, boolean blnTrade,
boolean blnOutput )
{
// init profit dispersal
dblPD = 0.0;
Hashtable UDC = UDCnet.getNetwork();
```

```
for ( int intServer = 1; intServer <= UDC.size(); intServer++ )
{
UDC_Server server = (UDC_Server) UDC.get( "S" + intServer );
// retrieve server buyer and seller agents
ZIP_Agent agtSERVbuyer = server.agtSERVbuyer;
ZIP_Agent agtSERVseller = server.agtSERVseller;
// calculate difference between actual gain and theoretical gain
dblDiff = (( agtSERVbuyer.dblAGNTactualgain ) -
( agtSERVbuyer.dblAGNTtheorgain ));
// square this difference
dblPD += ( dblDiff * dblDiff );
// calculate difference between actual gain and theoretical gain
dblDiff = (( agtSERVseller.dblAGNTactualgain ) -
( agtSERVseller.dblAGNTtheorgain ));
// square this difference
dblPD += ( dblDiff * dblDiff );
}
// calculate profit dispersal
dblProfitDisp = Math.sqrt( ( 1 / ( (double) ( intNoBuyers +
intNoSellers ) ) ) * dblPD );
if ( blnOutput )
{
//psOUT.println( "MBC_SIM: Profit Dispersal = " + dblProfitDisp );
}
// update 'day' stats if a trade occured
if ( blnTrade == true )
{
DayData[intTimeStep-1].data_day_update( intTradeNo, dblPriceSum,
dblAlpha, dblProfitDisp, dblEfficiency, dblPriceDiffSum,
intNoTrades );
}
}
// ---------------------------------------------------------------------
// TRADE_OUTPUT: output trade statistics
public void TRADE_output( int intTimeSteps, int[] intMaxTrades,
PrintStream psTRA, int intGen )
{
DecimalFormat df = new DecimalFormat( "0.00" );
if ( ( ( intGen % 50 ) == 0 ) || ( intGen == 1 ) )
{
// TRADE DATA OUTPUT: PRICE
psTRA.println( " PRICE (n=0 output only)" );
for ( int intTS = 0; intTS < intTimeSteps; intTS++ )
{
double dblDGX = ( 1.0 / ( (double) intMaxTrades[intTS] ) );
double dblGX = ( intTS + 1 );
double dblSum = DayData[intTS].dblSTAT_quant.dblSum;
for ( int intTrade = 0; intTrade < dblSum; intTrade++ )
{
if ( TradeData[intTS][intTrade].dblTDATdeal_price >= 0.0 )
{
psTRA.println( df.format( dblGX ) + " "
+ TradeData[intTS][intTrade].dblTDATdeal_price );
}
dblGX += dblDGX;
}
}
// TRADE DATA OUTPUT: ACTUAL EQUILIBRIUM PRICE
psTRA.println( " ACTUAL EQUILIBRIUM PRICE (n=0 output only)" );
for ( int intTS = 0; intTS < intTimeSteps; intTS++ )
{
double dblDGX = ( 1.0 / ( (double) intMaxTrades[intTS] ) );
double dblGX = ( intTS + 1 );
double dblSum = DayData[intTS].dblSTAT_quant.dblSum;
for ( int intTrade = 0; intTrade < dblSum; intTrade++ )
{
if ( TradeData[intTS][intTrade].intTDATactual_eq_quant
!= NULL_EQ )
```

```
{
if ( TradeData[intTS][intTrade].dblTDATactual_eq_price
> 0.0 )
{
psTRA.println( df.format( dblGX ) + " " +
df.format(
TradeData[intTS][intTrade].dblTDATactual_eq_price ) );
}
}
dblGX += dblDGX;
}
}
// (VIS) TRADE DATA OUTPUT: THEORETICAL EQUILIBRIUM PRICE
psTRA.println( " THEORETICAL EQUILIBRIUM PRICE "
+ "(n=0 output only)" );
for ( int intTS = 0; intTS < intTimeSteps; intTS++ )
{
double dblDGX = ( 1.0 / ( (double) intMaxTrades[intTS] ) );
double dblGX = ( intTS + 1 );
double dblSum = DayData[intTS].dblSTAT_quant.dblSum;
for ( int intTrade = 0; intTrade < dblSum; intTrade++ )
{
if ( TradeData[intTS][intTrade].intTDATtheor_eq_quant
!= NULL_EQ )
{
if ( TradeData[intTS][intTrade].dblTDATtheor_eq_price
> 0.0 )
{
psTRA.println( df.format( dblGX ) + " " +
df.format(
TradeData[intTS][intTrade].dblTDATtheor_eq_price ) );
}
}
dblGX += dblDGX;
}
}
}
}
// ---------------------------------------------------------------------
// DAY_OUTPUT: output day statistics (mean, +/- s.d.)
public void DAY_output( int intTimeSteps, PrintStream psDAY,
boolean[] blnTrade, int intGen )
{
ZIP_Data_Day ddVIS = new ZIP_Data_Day();
// PRICE
for ( int intTS = 0; intTS < intTimeSteps; intTS++ )
{
if ( blnTrade[intTS] == true )
{
TMPdblSum[intTS] = DayData[intTS].dblSTAT_price.dblSum;
TMPdblSumSq[intTS] = DayData[intTS].dblSTAT_price.dblSumSq;
TMPintN[intTS] = DayData[intTS].dblSTAT_price.intN;
}
}
ddVIS.data_day_stats( "PRICE", TMPdblSum, TMPdblSumSq, TMPintN,
intTimeSteps, psDAY, blnTrade, intGen );
// SMITH'S ALPHA
for ( int intTS = 0; intTS < intTimeSteps; intTS++ )
{
TMPdblSum[intTS] = DayData[intTS].dblSTAT_alpha.dblSum;
TMPdblSumSq[intTS] = DayData[intTS].dblSTAT_alpha.dblSumSq;
TMPintN[intTS] = DayData[intTS].dblSTAT_alpha.intN;
}
ddVIS.data_day_stats( "ALPHA", TMPdblSum, TMPdblSumSq, TMPintN,
intTimeSteps, psDAY, blnTrade, intGen );
// ---------------------------------------------------------------------
// (GA) CALCULATE FITNESS USING AVERAGE ALPHA PARAMETER
dblTrialFitness = ddVIS.data_day_alpha_fitness( intTimeSteps,
TMPdblSum, TMPintN, intNoTrades );
// ---------------------------------------------------------------------
```

```
// EFFICIENCY
for ( int intTS = 0; intTS < intTimeSteps; intTS++ )
{
TMPdblSum[intTS] = DayData[intTS].dblSTAT_effic.dblSum;
TMPdblSumSq[intTS] = DayData[intTS].dblSTAT_effic.dblSumSq;
TMPintN[intTS] = DayData[intTS].dblSTAT_effic.intN;
}
ddVIS.data_day_stats( "EFFICIENCY", TMPdblSum, TMPdblSumSq,
TMPintN, intTimeSteps, psDAY, blnTrade, intGen );
// QUANTITY
for ( int intTS = 0; intTS < intTimeSteps; intTS++ )
{
TMPdblSum[intTS] = DayData[intTS].dblSTAT_quant.dblSum;
TMPdblSumSq[intTS] = DayData[intTS].dblSTAT_quant.dblSumSq;
TMPintN[intTS] = DayData[intTS].dblSTAT_quant.intN;
}
ddVIS.data_day_stats( "QUANTITY", TMPdblSum, TMPdblSumSq,
TMPintN, intTimeSteps, psDAY, blnTrade, intGen );
// PROFIT DISPERSAL
for ( int intTS = 0; intTS < intTimeSteps; intTS++ )
{
TMPdblSum[intTS] = DayData[intTS].dblSTAT_pdisp.dblSum;
TMPdblSumSq[intTS] = DayData[intTS].dblSTAT_pdisp.dblSumSq;
TMPintN[intTS] = DayData[intTS].dblSTAT_pdisp.intN;
}
ddVIS.data_day_stats( "PROFIT DISPERSAL", TMPdblSum, TMPdblSumSq,
TMPintN, intTimeSteps, psDAY, blnTrade, intGen );
// VOLATILITY
for ( int intTS = 0; intTS < intTimeSteps; intTS++ )
{
TMPdblSum[intTS] = DayData[intTS].dblSTAT_volty.dblSum;
TMPdblSumSq[intTS] = DayData[intTS].dblSTAT_volty.dblSumSq;
TMPintN[intTS] = DayData[intTS].dblSTAT_volty.intN;
}
ddVIS.data_day_stats( "VOLATILITY", TMPdblSum, TMPdblSumSq, TMPintN,
intTimeSteps, psDAY, blnTrade, intGen );
}
// ---------------------------------------------------------------------
}
```

C.2.4 UDC_Network.java

Description: A Hashtable containing UDC_Server objects constituting the UDC network.

```
// Class: UDC_Network.java
import java.util.Hashtable;
public class UDC_Network extends Hashtable
{
// ---------------------------------------------------------------------
// return the UDC_Network
public Hashtable getNetwork()
{
return this;
}
// ---------------------------------------------------------------------
// add UDC_Server to Hashtable, using server ID as key
public void put( UDC_Server server )
{
put( server.strSERVid, server );
}
// ---------------------------------------------------------------------
}
```

C.2.5 UDC_Server.java

Description: An object constituting a server or node on the UDC network.

```
// Class: UDC_Server.java
import java.io.PrintStream;
import java.lang.Thread;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Random;
import java.util.Vector;
public class UDC_Server extends Thread implements ZIP_Constants
{
// ---------------------------------------------------------------------
// SERVER OBJECTS / PARAMETERS
// id
public String strSERVid;
// total resource units
public int intSERVtotal_ru;
// available resource units
public int intSERVavail_ru;
// the state of this server
public int intSERVstate;
// UDC_Task objects currently running (cannot be more than
// intSERVtotal_ru tasks)
public Vector vctSERVtasks;
// UDC_Server objects connected to
public Vector vctSERVservers;
// sells server resource units
public ZIP_Agent agtSERVseller;
// purchases other server resource units
public ZIP_Agent agtSERVbuyer;
// UDC_Tasks currently waiting in the local market
public Vector vctSERVmarket;
// ---------------------------------------------------------------------
// SERVER MBC PARAMETERS
// probability of a randomly configured new task appearing in the
// servers' local task market
private static double dblSERVtask_prob = 0.1;
// number of timesteps waiting for all willing flags after which the
// auction is terminated
private static int intSERVhalt_auction = 15;
// ---------------------------------------------------------------------
// SERVER MBC OBJECTS / MISC PARAMETERS
// SERVER STATES
private static int FREE = 0;
private static int PENDING_ACCEPTANCE = 1;
private static int WAITING_ACCEPTANCE = 2;
private static int WAITING_AUCTION = 3;
// a market task from this server put up for auction
public UDC_Task marketTask;
// a local server running task requested for auction
public UDC_Task localTask;
// a local server market task requested for auction
public UDC_Task bidTask;
// local server that has requested an auction
public UDC_Server SERVbid_server;
// the local server market task agent bid price
public double dblSERVbid_price;
// flag indicating if this server has returned whether willing to
// partake in auction or whether has returned an offer price
public boolean blnSERVret_willing;
// flag indicating if the server is willing to partake in requested
// auction
public boolean blnSERVwilling;
// flag indicating if the server is eligable to shift a currently
// running task in order to partake in a locally requested auction
public boolean blnEnableShifting;
// flag indicating if the server is eligable to shift one task
```

```
public boolean blnEnableOneShift;
// offer price of this server in response to local auction request
public double dblSERVofferprice;
// indicates if locally requested auction to be a bid or offer deal
public boolean blnDealType;
// used for storing surplus value
private double dblSurplus;
// counters record number of timesteps server is waiting for task put
// up from local market (1) or for a task currently running, to
// offload (2)
public int intNoStepsWaiting1;
public int intNoStepsWaiting2;
// used for randomly selecting a single server to task shift
public int intSERVid1;
// indicates whether server is waiting for local server responses for
// an auction based on local market task or currently running task
public boolean blnWillingWaitType;
// used to temporarily halt an auction if not all local server
// responses recieved or need to carry out n or more local server
// task shifts
public boolean blnStopAuction;
// count indicating the number of tasks processed
public int intSERVjprsd;
// count indicating the number of tasks allocated
public int intSERValloc;
// close deal by selecting local server randomly or that offering the
// best deal
private boolean blnCompetitive = false;
// indicates if an auction occured at this server, at this timestep
public boolean blnAuctionOccured;
// indicates if a trade occured at this server, at this timestep
public boolean blnTradeOccured;
// current auction status (preset to no deal)
private int intAuctionStatus = NO_DEAL;
// probability of auction being bid or offer based
public double dblQS;
// ---------------------------------------------------------------------
// SERVER INITIALISATION CONSTRUCTOR
public UDC_Server( String TMPstrSERVid, int TMPintSERVtotal_ru )
{
// initialise server data structures
strSERVid = TMPstrSERVid;
intSERVtotal_ru = TMPintSERVtotal_ru;
intSERVavail_ru = TMPintSERVtotal_ru;
vctSERVtasks = new Vector();
vctSERVservers = new Vector();
agtSERVseller = new ZIP_Agent();
agtSERVbuyer = new ZIP_Agent();
vctSERVmarket = new Vector();
// initialise server to 'free' state
intSERVstate = FREE;
bidTask = new UDC_Task( "null", 0, 0, 0.0 );
marketTask = new UDC_Task( "null", 0, 0, 0.0 );
blnSERVret_willing = false;
blnSERVwilling = false;
blnWillingWaitType = false;
intNoStepsWaiting1 = 0;
intNoStepsWaiting2 = 0;
blnStopAuction = false;
intSERVid1 = 0;
blnAuctionOccured = false;
blnTradeOccured = false;
dblSERVofferprice = 0.0;
intSERVjprsd = 0;
dblQS = 0.5;
//blnEnableShifting = true;
}
// ---------------------------------------------------------------------
// UPDATE SERVER STATE
public int updateServer( int intTimeStep, int intTASKid,
```

```
boolean blnOutput, ZIP_Stats stats, Random random, PrintStream psOUT,
PrintStream psSDO, double dblAGNT_lr_L, double dblAGNT_lr_H,
double dblAGNT_mm_L, double dblAGNT_mm_H, double dblAGNT_pf_L,
double dblAGNT_pf_H, double dblAGNT_tr_R, double dblAGNT_tr_A,
double dblPrice_0, double dblMaxSurplus, UDC_Network UDCnet,
ZIP_SD_Vis SDVIS, int intVISno_timesteps )
{
// number of buyers
int intNoBuyers;
// number of sellers
int intNoSellers;
// equilibrium quantity
int intEqQuantity = 0;
// (used in calling ZIP_SD_Vis.SDVIS_sup_dem method)
int intDummy = 0;
// equilibrium price
double dblEqPrice = 0.0;
// current actual maximum surplus
double dblCurrMaxSurp = 0.0;
// used in NYSE rules
double dblBestOffer = 0.0;
// used in NYSE rules
double dblBestBid = 0.0;
// price of bid / offer
double dblPrice = 0.0;
// used in calling ZIP_SD_Vis.SDVIS_sup_dem method
double dblDummy1 = 0.0;
// used in calling ZIP_SD_Vis.SDVIS_sup_dem method
double dblDummy2 = 0.0;
blnTradeOccured = false;
// ------------------------------------------------------------------
// DO THINGS WHICH HAPPEN REGARDLESS AT EACH TIMESTEP:
// decrement by one the intTASKduration value all of the UDC_Tasks
// (if any) currently running on the UDC_Server
decTaskDurations( blnOutput );
// with random probability, create a new task in the local market
//if ( random.nextDouble() < dblSERVtask_prob )
//{
if (( strSERVid.equals( "S45" )) || ( strSERVid.equals( "S55" )))
{
//for ( int i = 0; i < 10; i++ )
//{
// task identification
String strTASKid = "T" + intTASKid;
// randomly determine a task duration (range [3,10])
int intTASKduration = 500;
// ( random.nextInt( 25 ) ) + 3 OR 250
// randomly determine a task resource requirement (range [1,6])
int intTASKload = 1;
// or ( random.nextInt( 5 ) ) + 1 OR 1
// randomly determine funds to spend on resources
// (range[0.75,3.25] - to represent demand curve values)
double dblTASKfunds = ( random.nextDouble() * 2.5 ) + 0.75;
// OR ( random.nextDouble() * 2.5 ) + 0.75; FOR range[0.75,3.25]
// instantiate the UDC_Task object
UDC_Task newTask = new UDC_Task( strTASKid, intTASKduration,
intTASKload, dblTASKfunds );
if ( blnOutput )
{
psOUT.println( "MBC_SIM: new TASK " + strTASKid +
" entered local market on SERVER " + strSERVid );
psOUT.println();
}
newTask.taskBuyer.dblAGNT_lr_L = dblAGNT_lr_L;
newTask.taskBuyer.dblAGNT_lr_H = dblAGNT_lr_H;
newTask.taskBuyer.dblAGNT_mm_L = dblAGNT_mm_L;
newTask.taskBuyer.dblAGNT_mm_H = dblAGNT_mm_H;
newTask.taskBuyer.dblAGNT_pf_L = dblAGNT_pf_L;
newTask.taskBuyer.dblAGNT_pf_H = dblAGNT_pf_H;
```

```
newTask.taskBuyer.dblAGNT_tr_R = dblAGNT_tr_R;
newTask.taskBuyer.dblAGNT_tr_A = dblAGNT_tr_A;
newTask.taskBuyer.AGNT_init_buyer( ( 1000 + intTASKid ),
random, blnOutput );
// SET TASK BUYER AGENT QUANTITY AND SET ACTIVE
newTask.taskBuyer.dblAGNTactualgain = 0.0;
// SET TASK BUYER AGENT LIMIT PRICE
// the buyer agent will not pay more than what the agent has
// in funds (defines the priority of the task)
newTask.taskBuyer.dblAGNTlimit = dblTASKfunds;
newTask.taskBuyer.AGNT_calcPrice();
// add the task to the local market
addTaskToMarket( newTask );
intTASKid++;
//}
}
// ---------------------------------------------------------------
// SERVER STATE ACTIONS (FOR THIS TIMESTEP)
// ---------------------------------------------------------------
if ( ( intTimeStep % 60 ) == 0 )
{
//intSERVstate = FREE;
}
if ( intSERVstate == FREE )
{
// IF ANY TASKS WAITING IN LOCAL SERVER MARKET (ELSE REMAIN IN A
// FREE STATE)
if ( vctSERVmarket.size() > 0 )
{
// grab a task from the local market
Enumeration enumTasks = vctSERVmarket.elements();
marketTask = (UDC_Task) enumTasks.nextElement();
// get the bid price price for the task buyer agent
double dblBidPrice = marketTask.getPrice( marketTask.taskBuyer,
marketTask.strTASKid, blnOutput );
if ( blnOutput )
{
psOUT.println();
psOUT.println( "SERVER " + strSERVid + ": TASK "
+ marketTask.strTASKid + " in local market to be put up "
+ "for auction, task buyer agent price: " + dblBidPrice );
psOUT.println();
}
int intNoAbleNeighbours = 0;
// randomly determine whether this auction will be closed via
// a bid or an offer
double dblDealType = random.nextDouble();
if ( dblDealType > dblQS )
{
// auction to be closed via a BID
blnDealType = true;
}
else
{
// auction to be closed via an OFFER
blnDealType = false;
}
Enumeration enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
// IF NEIGHBOURING SERVER IN FREE STATE
if ( neighServer.intSERVstate == FREE )
{
// SET THE BID TASK AND BID PRICE ON NEIGHBOURING SERVER
neighServer.bidTask = marketTask;
neighServer.SERVbid_server = this;
neighServer.dblSERVbid_price = dblBidPrice;
```

```
if ( dblDealType > dblQS )
{
// auction to be closed via a BID
neighServer.blnDealType = true;
}
else
{
// auction to be closed via an OFFER
neighServer.blnDealType = false;
}
// change state of neighbouring server
neighServer.intSERVstate = PENDING_ACCEPTANCE;
intNoAbleNeighbours++;
}
}
// IF NO NEIGHBOURING SERVERS ABLE TO PARTAKE REMAIN IN A FREE
// STATE
if ( intNoAbleNeighbours == 0 )
{
intSERVstate = FREE;
blnSERVwilling = false;
blnSERVret_willing = true;
if ( blnOutput )
{
psOUT.println( "SERVER " + strSERVid +
": No neighbours in a free state to participate " +
" in auction for TASK: " + marketTask.strTASKid +
". Task left in local market" );
}
}
// ELSE THERE COULD BE WILLING SERVERS
else
{
// PUT SERVER INTO STATE WAITING FOR WILLING FLAGS
intSERVstate = WAITING_ACCEPTANCE;
blnSERVret_willing = false;
blnWillingWaitType = true;
}
}
}
// ---------------------------------------------------------------------
// WAITING FOR NEIGHBOURING SERVERS WILLING FLAGS
else if ( intSERVstate == WAITING_ACCEPTANCE )
{
// IF FOR A BID PUT OUT FOR TASK CURRENTLY IN *LOCAL SERVER MARKET*
if ( blnWillingWaitType == true )
{
boolean blnAllRecieved = true;
Enumeration enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server) enumServers.nextElement();
// if neighbouring server recieved the bid for market task from
// this server
if ( marketTask.strTASKid.equals( neighServer.bidTask.strTASKid ) )
{
if ( neighServer.blnSERVret_willing == false )
{
// server hasn't returned willing flag so yet to recieve all
// willing flags
blnAllRecieved = false;
}
}
}
// initialise willing list structure and indices count
String[] strWillingList = new String[MAX_AGENTS];
int intNoWilling = 0;
int intNoFails = 0;
// ---------------------------------------------------------------------
// IF NOT YET RECIEVED ALL NEIGHBOURING SERVERS WILLING FLAGS
if ( blnAllRecieved == false )
{
```

```
// remain in same state
intSERVstate = WAITING_ACCEPTANCE;
// increment number of timesteps waiting for willing flags
intNoStepsWaiting1++;
// if the number of timesteps waiting now above limit
if ( intNoStepsWaiting1 > intSERVhalt_auction )
{
if ( blnOutput )
{
psOUT.println( "SERVER: " + strSERVid +
" has been waiting for all willing flags for more than "
+ intSERVhalt_auction + " timesteps (at timestep "
+ intTimeStep + ")" );
}
// OPTION: add in three lines below and neigh. server line
// below to have this server return as not willing in
// original auction if waited more than n timesteps
intSERVstate = FREE;
blnSERVwilling = false;
blnSERVret_willing = true;
// set all neighbouring servers involved in auction in a free
// state
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
// if neighbouring server recieved the bid for local task
// from this server
if ( marketTask.strTASKid.equals(
neighServer.bidTask.strTASKid ) )
{
neighServer.intSERVstate = FREE;
}
}
}
}
// ------------------------------------------------------------------
// ELSE HAVE RECIEVED ALL NEIGHBOURING SERVERS WILLING FLAGS
// GO AHEAD WITH AUCTION
else
{
// reset stop auction flag
if ( blnStopAuction == true )
{
blnStopAuction = false;
}
// reset number of timesteps waiting count
intNoStepsWaiting1 = 0;
// reset auction status flag to NO_DEAL
intAuctionStatus = NO_DEAL;
// do the deal based on a BID
if ( blnDealType == true )
{
// get willing servers
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
// if neighbouring server recieved the bid for market
// task from this server
if ( marketTask.strTASKid.equals(
neighServer.bidTask.strTASKid ) )
{
// if waiting for the auction (i.e. is willing)
if ( neighServer.intSERVstate == WAITING_AUCTION )
{
// (check again whether it has enough available capacity)
if ( neighServer.intSERVavail_ru >=
```

```
neighServer.bidTask.intTASKload )
{
// ADD SERVER ID TO THE WILLING LIST
strWillingList[intNoWilling] = neighServer.strSERVid;
intNoWilling++;
}
}
}
}
// now check whether THIS server is willing
// if server has enough available capacity
if ( intSERVavail_ru >= marketTask.intTASKload )
{
blnSERVwilling = isWilling( marketTask.taskBuyer.dblAGNTprice,
blnOutput );
if ( blnSERVwilling == true )
{
// ADD SERVER ID TO THE WILLING LIST
strWillingList[intNoWilling] = strSERVid;
intNoWilling++;
}
}
}
// do the deal based on an OFFER
else
{
// determine which servers to add to the willing list based
// on those giving offer prices that are accepted by the task
// buyer agent
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
// if neighbouring server recieved the bid for market task
// from this server
if ( marketTask.strTASKid.equals(
neighServer.bidTask.strTASKid ) )
{
// if waiting for the auction (i.e. is willing)
//if ( neighServer.intSERVstate == WAITING_AUCTION )
//{
// (check again whether it has enough available capacity)
if ( neighServer.intSERVavail_ru >=
neighServer.bidTask.intTASKload )
{
// ADD SERVER ID TO THE WILLING BASED ON OFFER PRICE
// get the server seller offer price
double dblOfferPrice = neighServer.dblSERVofferprice;
   if ( dblOfferPrice != 0.0 )
   {
// get whether this server's buyer agent is willing
// to deal
boolean blnBuyWilling = isBuyWilling( dblOfferPrice,
blnOutput );
if ( blnBuyWilling == true )
{
strWillingList[intNoWilling] =
neighServer.strSERVid;
intNoWilling++;
}
}
}
//}
}
}
// now check whether THIS servers offer price is 'good enough'
// if server has enough available capacity
if ( intSERVavail_ru >= marketTask.intTASKload )
{
```

```
// get the server seller offer price
double dblOfferPrice = dblSERVofferprice;
// get whether this server's buyer agent is willing to deal
boolean blnBuyWilling = isBuyWilling( dblOfferPrice,
blnOutput );
if ( blnBuyWilling == true )
{
strWillingList[intNoWilling] = strSERVid;
intNoWilling++;
}
}
}
// ---------------------------------------------------------------------
// GET NUMBER OF BUYER AND SELLER AGENTS
// number of buyer agents equal to the one on the new task
intNoBuyers = 1;
// number of sellers equal to the number of willing servers
intNoSellers = intNoWilling;
// FIND THE THEORETICAL EQUILIBRIUM PRICE
SDVIS.SDVIS_sup_dem( intNoSellers, intNoBuyers, UDCnet, dblEqPrice,
intEqQuantity, dblCurrMaxSurp, EQ_THEORY, 0.0, blnOutput, psSDO );
dblEqPrice = SDVIS.dblEqPrice;
intEqQuantity = SDVIS.intIQuant;
dblCurrMaxSurp = SDVIS.dblSurplus;
// (use to update day and trade statistics in place of initialisation
// theoretical equilibrium price)
double TdblEqPrice = SDVIS.dblEqPrice;
int TintEqQuantity = SDVIS.intIQuant;
double TdblCurrMaxSurp = SDVIS.dblSurplus;
// FIND THE ACTUAL EQUILIBRIUM PRICE
SDVIS.SDVIS_sup_dem( intNoSellers, intNoBuyers, UDCnet, dblEqPrice,
intEqQuantity, dblCurrMaxSurp, EQ_ACTUAL, 0.0, blnOutput, psSDO );
dblEqPrice = SDVIS.dblEqPrice;
intEqQuantity = SDVIS.intIQuant;
dblCurrMaxSurp = SDVIS.dblSurplus;
// ---------------------------------------------------------------------
// XFIG: ...print a figure of the actual supply and demand curves...
SDVIS.SDVIS_sup_dem( intNoSellers, intNoBuyers, UDCnet, dblEqPrice,
intEqQuantity, dblCurrMaxSurp, EQ_THEORY, 0.0, blnOutput, psSDO );
dblDummy1 = SDVIS.dblEqPrice;
intDummy = SDVIS.intIQuant;
dblDummy2 = SDVIS.dblSurplus;
// ---------------------------------------------------------------------
int intCount = 0;
// IF ANY WILLING SERVERS THEN ENTER INTO A DEAL
if ( intNoWilling > 0 )
{
intAuctionStatus = DEAL;
}
else
{
// NO SERVERS WILLING SO ENABLE A TASK SHIFT
if ( blnEnableOneShift == false )
{
// get the integer id of this server
intSERVid1 = Integer.parseInt( strSERVid.substring( 1 ) );
boolean blnTemp = false;
// randomly select a server to try and offload a job
// (while *not* this server)
while ( intSERVid1 == Integer.parseInt(
strSERVid.substring( 1 ) ) )
{
// while equal to a neighbouring server not involved in
// this auction
while( blnTemp == false )
{
// RANDOMLY SELECT ONE SERVER TO TRY AND OFFLOAD A JOB
// (but have to select server that part of *this*
```

```
// auction only)
random.setSeed( (int) ( Math.random() * 100 ) );
intSERVid1 = (int) ( random.nextDouble() *
vctSERVservers.size() );
// get the integer id of the of the randomly selected
// server
intSERVid1 += Integer.parseInt(
strSERVid.substring( 1 ) ) - 5;
// if neighbouring server recieved the bid for market
// task from this server
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
if ( intSERVid1 == Integer.parseInt(
neighServer.strSERVid.substring( 1 )) )
{
if ( marketTask.strTASKid.equals(
neighServer.bidTask.strTASKid ) )
{
// randomly selected server is equal to server
// involved in this auction
blnTemp = true;
}
}
}
intCount++;
if ( intCount > 3 )
{
blnTemp = true;
}
}
}
// ENUMERATE THROUGH TO THIS RANDOMLY SELECTED SERVER
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
if ( neighServer.strSERVid.equals( "S" + intSERVid1 ) )
{
// ENABLE SHIFTING IN THIS SINGLE RANDOMLY SELECTED
// SERVER
neighServer.blnEnableShifting = true;
// IMPORTANT: (just removing this line removes all task
// shifting)
neighServer.intSERVstate = PENDING_ACCEPTANCE;
}
}
// turn on boolean flag to enable task shifting
blnEnableOneShift = true;
// (TEMPORARILY) HALT THIS AUCTION TO ALLOW FOR THE TASK SHIFT
blnStopAuction = true;
}
}
// ---------------------------------------------------------------------
// IF AUCTION STILL GOING AHEAD, carry on
if ( blnStopAuction == false )
{
blnAuctionOccured = true;
ZIP_Agent CHOSEN_agtSERVseller = new ZIP_Agent();
// if there is at least one willing server
if ( intAuctionStatus == DEAL )
{
int intTemp = (int) ( random.nextDouble() * intNoWilling );
String CHOSEN_strSERVid = strWillingList[intTemp];
// RANDOMLY SELECT A WILLING SERVER SELLER FOR THIS TASK
if ( blnCompetitive == false )
{
```

```
// if the randomly selected seller agent is on the LOCAL
// SERVER
if ( CHOSEN_strSERVid.equals( strSERVid ) )
{
CHOSEN_agtSERVseller = agtSERVseller;
}
// else the randomly selected seller agent is on a
// NEIGHBOURING SERVER
else
{
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
if ( CHOSEN_strSERVid.equals( neighServer.strSERVid ) )
{
CHOSEN_agtSERVseller = neighServer.agtSERVseller;
}
}
}
}
// or out of the willing servers SELECT THE ONE THAT IS GIVES
// THE BEST REWARD for the required resource units
else
{
double dblLargeReward = 0.0;
// loop through all servers in willing list
for ( int intWillServ = 0; intWillServ < intNoWilling;
intWillServ++ )
{
// retrieve server id from willing list
String TMPstrSERVid = strWillingList[intWillServ];
// if the randomly selected seller agent is on the LOCAL
// SERVER
if ( TMPstrSERVid.equals( strSERVid ) )
{
CHOSEN_agtSERVseller = agtSERVseller;
double dblReward = reward( CHOSEN_agtSERVseller,
dblPrice );
// select server giving the best reward
if ( dblReward > dblLargeReward )
{
dblLargeReward = dblReward;
CHOSEN_strSERVid = strSERVid;
CHOSEN_agtSERVseller = agtSERVseller;
}
}
// else the randomly selected seller agent is on a
// NEIGHBOURING SERVER
else
{
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
if ( TMPstrSERVid.equals( neighServer.strSERVid ) )
{
CHOSEN_agtSERVseller = neighServer.agtSERVseller;
double dblReward = reward( CHOSEN_agtSERVseller,
dblPrice );
// select server giving the best reward
if ( dblReward > dblLargeReward )
{
dblLargeReward = dblReward;
CHOSEN_strSERVid = neighServer.strSERVid;
CHOSEN_agtSERVseller =
neighServer.agtSERVseller;
}
}
```

```
}
}
}
}
if ( blnOutput )
{
// if deal based on the task buyers BID
if ( blnDealType == true )
{
System.out.println( "BUYER FOR TASK " + marketTask.strTASKid +
" BUYS FROM SELLER on server " + CHOSEN_strSERVid +
" (reward = " + reward( CHOSEN_agtSERVseller,
marketTask.taskBuyer.dblAGNTprice ) + ")" );
}
// else if deal based on the server sellers OFFER
else
{
System.out.println( "SELLER FOR SERVER " + CHOSEN_strSERVid +
" SELLS TO BUYER on TASK " + marketTask.strTASKid +
" (reward = " + reward( CHOSEN_agtSERVseller,
marketTask.taskBuyer.dblAGNTprice ) + ")" );
}
}
intSERValloc++;
// UPDATE TRADE STATISTICS
stats.TRADE_update_the_eq( intTimeStep, TdblEqPrice, TintEqQuantity );
stats.TRADE_updade_act_eq( intTimeStep, dblEqPrice, intEqQuantity );
// call methods to record trade price and deal type
stats.TRADE_deal_type_price( intTimeStep,
marketTask.taskBuyer.dblAGNTprice, BID );
// ---------------------------------------------------------------
// UPDATE TRADING STRATEGIES
// now a trade has taken place, update the trading strategy of the
// task buyer agent, the server buyer agents, and the server seller
// agents (whether they participated in the auction or not)
marketTask.taskBuyer.AGNT_shout_update_buyer( marketTask.strTASKid,
blnDealType, intAuctionStatus, marketTask.taskBuyer.dblAGNTprice,
random, blnOutput );
// update trading strategies: local server
agtSERVbuyer.AGNT_shout_update_buyer( strSERVid, blnDealType,
intAuctionStatus, marketTask.taskBuyer.dblAGNTprice, random,
blnOutput );
agtSERVseller.AGNT_shout_update_seller( strSERVid, blnDealType,
intAuctionStatus, marketTask.taskBuyer.dblAGNTprice, random,
intTimeStep, blnOutput );
// update trading strategies: neighbouring servers
enumServers = vctSERVservers.elements();
// enumerate through all neighbouring servers
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server) enumServers.nextElement();
neighServer.agtSERVbuyer.AGNT_shout_update_buyer(
neighServer.strSERVid, blnDealType, intAuctionStatus,
marketTask.taskBuyer.dblAGNTprice, random, blnOutput );
neighServer.agtSERVseller.AGNT_shout_update_seller(
neighServer.strSERVid, blnDealType, intAuctionStatus,
marketTask.taskBuyer.dblAGNTprice, random, intTimeStep,
blnOutput );
}
// ---------------------------------------------------------------
// UPDATE BANK ACCOUNTS AND ADD THE TASK TO THE SERVER
// of the task buyer agent and the server seller agent
bank( marketTask.taskBuyer, marketTask.taskBuyer.dblAGNTprice,
blnOutput );
removeTaskFromMarket( marketTask );
// if the randomly selected seller agent was on the LOCAL SERVER
if ( CHOSEN_strSERVid.equals( strSERVid ) )
{
```

```
bank( agtSERVseller, marketTask.taskBuyer.dblAGNTprice,
blnOutput );
addUDC_Task( marketTask );
}
// else the randomly selected seller agent was a NEIGHBOURING SERVER
else
{
enumServers = vctSERVservers.elements();
// iterate through the neighbouring servers
while ( enumServers.hasMoreElements() )
{
// get the neighbouring server object
UDC_Server neighServer = (UDC_Server) enumServers.nextElement();
if ( CHOSEN_strSERVid.equals( neighServer.strSERVid ) )
{
neighServer.bank( neighServer.agtSERVseller,
marketTask.taskBuyer.dblAGNTprice, blnOutput );
neighServer.addUDC_Task( marketTask );
}
}
}
// ------------------------------------------------------------------
// record what the server paid for the task
marketTask.dblTASKworth = marketTask.taskBuyer.dblAGNTprice;
blnTradeOccured = true;
}
// ------------------------------------------------------------------
// IF NO DEAL (TASK REMAINS IN LOCAL SERVER MARKET)
else
{
// increment number of fails count (if not already set at max fails)
if ( intNoFails < MAX_FAILS )
{
intNoFails++;
}
if ( blnOutput )
{
psOUT.println(
"No servers willing to sell resources (fails = " + intNoFails +
")" );
psOUT.println();
}
// record fact there was no deal
stats.TRADE_no_deal( intTimeStep );
// ------------------------------------------------------------------
// UPDATE TRADING STRATEGIES
// now a trade has taken place, update the trading strategy of the
// task buyer agent, the server buyer agents, and the server seller
// agents
marketTask.taskBuyer.AGNT_shout_update_buyer(
marketTask.strTASKid, blnDealType, intAuctionStatus,
marketTask.taskBuyer.dblAGNTprice, random, blnOutput );
// LOCAL SERVER
agtSERVbuyer.AGNT_shout_update_buyer( strSERVid, blnDealType,
intAuctionStatus, marketTask.taskBuyer.dblAGNTprice, random,
blnOutput );
agtSERVseller.AGNT_shout_update_seller( strSERVid, blnDealType,
intAuctionStatus, marketTask.taskBuyer.dblAGNTprice, random,
intTimeStep, blnOutput );
// NEIGHBOURING SERVERS
enumServers = vctSERVservers.elements();
// iterate through the neighbouring servers
while ( enumServers.hasMoreElements() )
{
// get the neighbouring server object
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
neighServer.agtSERVbuyer.AGNT_shout_update_buyer(
neighServer.strSERVid, blnDealType, intAuctionStatus,
```

```
marketTask.taskBuyer.dblAGNTprice, random, blnOutput );
neighServer.agtSERVseller.AGNT_shout_update_seller(
neighServer.strSERVid, blnDealType, intAuctionStatus,
marketTask.taskBuyer.dblAGNTprice, random, intTimeStep,
blnOutput );
}
}
// ---------------------------------------------------------------------
// AUCTION COMPLETED: RETURN ALL SERVERS INVOLVED IN AUCTION TO A
// FREE STATE
intSERVstate = FREE;
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
// if neighbouring server recieved the bid for market
// task from this server
if ( marketTask.strTASKid.equals(
neighServer.bidTask.strTASKid ) )
{
// RETURN SERVER INVOLVED IN AUCTION TO A FREE STATE
neighServer.intSERVstate = FREE;
if ( neighServer.strSERVid.equals( "S" + intSERVid1 ) )
{
// reset enable shifting flags back to false
neighServer.blnEnableShifting = false;
blnEnableOneShift = false;
}
}
}
if ( intAuctionStatus == DEAL )
{
stats.STATS_calculate( intTimeStep, intAuctionStatus,
marketTask.taskBuyer.dblAGNTprice, TdblEqPrice,
dblSurplus, TdblCurrMaxSurp, blnOutput );
}
}
}
}
// ---------------------------------------------------------------------
// IF FOR A BID PUT OUT FOR TASK CURRENTLY RUNNING (in order that this
// server can partake in auction for a task put up *by another*
// neighbouring server)
else if ( blnWillingWaitType == false )
{
Enumeration enumServers = vctSERVservers.elements();
boolean blnAllRecieved = true;
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server) enumServers.nextElement();
// if neighbouring server recieved the bid for local task from
// this server
if ( localTask.strTASKid.equals( neighServer.bidTask.strTASKid ) )
{
if ( neighServer.blnSERVret_willing == false )
{
// server hasn't returned willing flag so yet to recieve all
// willing flags
blnAllRecieved = false;
}
}
}
// initialise willing list structure and indices count
String[] strWillingList = new String[MAX_AGENTS];
int intNoWilling = 0;
int intNoFails = 0;
// ---------------------------------------------------------------------
// IF NOT YET RECIEVED ALL NEIGHBOURING SERVERS WILLING FLAGS
if ( blnAllRecieved == false )
{
```

```
// remain in same state
intSERVstate = WAITING_ACCEPTANCE;
// increment number of timesteps waiting for willing flags
intNoStepsWaiting2++;
// if the number of timesteps waiting now above limit
if ( intNoStepsWaiting2 > intSERVhalt_auction )
{
psOUT.println( "SERVER: " + strSERVid +
" has been waiting for all willing flags for more than "
+ intSERVhalt_auction + " timesteps (at timestep "
+ intTimeStep + ")" );
// OPTION: add in three lines below and neigh. server line
// below to have this server return as not willing in original
// auction if waited more than n timesteps
intSERVstate = FREE;
blnSERVwilling = false;
blnSERVret_willing = true;
// set all neighbouring servers involved in auction in a free
// state
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
// if neighbouring server recieved the bid for local
// task from this server
if ( localTask.strTASKid.equals(
neighServer.bidTask.strTASKid ) )
{
neighServer.intSERVstate = FREE;
}
}
}
}
// ---------------------------------------------------------------
// ELSE HAVE RECIEVED ALL NEIGHBOURING SERVERS WILLING FLAGS
// (go ahead with auction)
else
{
// reset number of timesteps waiting count
intNoStepsWaiting2 = 0;
// reset auction status flag to NO_DEAL
int intAuctionStatus = NO_DEAL;
// do the deal based on a BID
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
// if neighbouring server recieved the bid for market task
// from this server
if ( localTask.strTASKid.equals(
neighServer.bidTask.strTASKid ) )
{
// if waiting for the auction (i.e. is willing)
if ( neighServer.intSERVstate == WAITING_AUCTION )
{
// (check again whether it has enough available
// capacity)
if ( neighServer.intSERVavail_ru >=
neighServer.bidTask.intTASKload )
{
// ADD SERVER ID TO THE WILLING LIST
strWillingList[intNoWilling] = neighServer.strSERVid;
intNoWilling++;
}
}
}
}
// (Nb. this server not willing because it is the server trying to
```

```
// offload task)
// ---------------------------------------------------------------------
// GET NUMBER OF BUYER AND SELLER AGENTS
// number of buyer agents equal to the one on the new task
intNoBuyers = 1;
// number of sellers equal to the number of willing servers
intNoSellers = intNoWilling;
// FIND THE THEORETICAL EQUILIBRIUM PRICE
SDVIS.SDVIS_sup_dem( intNoSellers, intNoBuyers, UDCnet, dblEqPrice,
intEqQuantity, dblCurrMaxSurp, EQ_THEORY, 0.0, blnOutput, psSD0 );
dblEqPrice = SDVIS.dblEqPrice;
intEqQuantity = SDVIS.intIQuant;
dblCurrMaxSurp = SDVIS.dblSurplus;
// (use to update day and trade statistics in place of initialisation
// theoretical equilibrium price)
double TdblEqPrice = SDVIS.dblEqPrice;
int TintEqQuantity = SDVIS.intIQuant;
double TdblCurrMaxSurp = SDVIS.dblSurplus;
// FIND THE ACTUAL EQUILIBRIUM PRICE
SDVIS.SDVIS_sup_dem( intNoSellers, intNoBuyers, UDCnet, dblEqPrice,
intEqQuantity, dblCurrMaxSurp, EQ_ACTUAL, 0.0, blnOutput, psSD0 );
dblEqPrice = SDVIS.dblEqPrice;
intEqQuantity = SDVIS.intIQuant;
dblCurrMaxSurp = SDVIS.dblSurplus;
// ---------------------------------------------------------------------
// XFIG: ...print a figure of the actual supply and demand curves...
SDVIS.SDVIS_sup_dem( intNoSellers, intNoBuyers, UDCnet, dblEqPrice,
intEqQuantity, dblCurrMaxSurp, EQ_THEORY, 0.0, blnOutput, psSD0 );
dblDummy1 = SDVIS.dblEqPrice;
intDummy = SDVIS.intIQuant;
dblDummy2 = SDVIS.dblSurplus;
// ---------------------------------------------------------------------
// IF ANY WILLING SERVERS THEN ENTER INTO A DEAL
if ( intNoWilling > 0 )
{
intAuctionStatus = DEAL;
}
else
{
// NO SERVERS WILLING SO ENABLE ANOTHER TASK SHIFT
// count number of neighbours involved in the auction
int intNeighbourCount = 0;
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
if ( localTask.strTASKid.equals(
neighServer.bidTask.strTASKid ) )
{
intNeighbourCount++;
}
}
// RANDOMLY SELECT ONE SERVER TO TRY AND OFFLOAD A JOB
int TMPintSERVid = (int) ( random.nextDouble() *
intNeighbourCount );
intNeighbourCount = 0;
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
if ( localTask.strTASKid.equals(
neighServer.bidTask.strTASKid ) )
{
intNeighbourCount++;
if ( TMPintSERVid == intNeighbourCount )
{
```

```
// ENABLE SHIFTING IN THIS SINGLE RANDOMLY SELECTED
// SERVER
neighServer.blnEnableShifting = true;
// (just removing this line removes all task shifting)
neighServer.intSERVstate = PENDING_ACCEPTANCE;
}
}
}
// turn on boolean flag to enable task shifting
blnEnableOneShift = true;
// (TEMPORARILY) HALT THIS AUCTION TO ALLOW FOR THE TASK SHIFT
blnStopAuction = true;
}
// ---------------------------------------------------------------------
// IF AUCTION STILL GOING AHEAD, carry on
if ( blnStopAuction == false )
{
blnAuctionOccured = true;
ZIP_Agent CHOSEN_agtSERVseller = new ZIP_Agent();
// if there is at least one willing server
if ( intAuctionStatus == DEAL )
{
// RANDOMLY SELECT A WILLING SERVER SELLER FOR THIS TASK
int intTemp = (int) ( random.nextDouble() * intNoWilling );
String CHOSEN_strSERVid = strWillingList[intTemp];
if ( blnCompetitive == false )
{
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
if ( CHOSEN_strSERVid.equals( neighServer.strSERVid ) )
{
CHOSEN_agtSERVseller = neighServer.agtSERVseller;
}
}
}
// or out of the willing servers SELECT THE ONE THAT IS GIVES
// THE BEST REWARD for the required resource units
else
{
double dblLargeReward = 0.0;
// loop through all servers in willing list
for ( int intWillServ = 0; intWillServ < intNoWilling;
intWillServ++ )
{
// retrieve server id from willing list
String TMPstrSERVid = strWillingList[intWillServ];
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server)
enumServers.nextElement();
if ( TMPstrSERVid.equals( neighServer.strSERVid ) )
{
CHOSEN_agtSERVseller = neighServer.agtSERVseller;
double dblReward = reward( CHOSEN_agtSERVseller,
dblPrice );
// select server giving the best reward
if ( dblReward > dblLargeReward )
{
dblLargeReward = dblReward;
CHOSEN_strSERVid = neighServer.strSERVid;
CHOSEN_agtSERVseller = neighServer.agtSERVseller;
}
}
}
}
}
```

```
if ( blnOutput )
{
psOUT.println( "BUYER on server " + strSERVid +
" FOR TASK " + localTask.strTASKid +
" BUYS FROM SELLER on server " + CHOSEN_strSERVid +
" (reward = " + reward( agtSERVseller,
agtSERVbuyer.dblAGNTprice ) + ")" );
}
intSERValloc++;
// UPDATE TRADE STATISTICS
stats.TRADE_update_the_eq( intTimeStep, TdblEqPrice,
TintEqQuantity );
stats.TRADE_updade_act_eq( intTimeStep, dblEqPrice,
intEqQuantity );
// call methods to record trade price and deal type
stats.TRADE_deal_type_price( intTimeStep,
agtSERVbuyer.dblAGNTprice, BID );
// ---------------------------------------------------------------
// UPDATE TRADING STRATEGIES
// now a trade has taken place, update the trading strategy of the task
// buyer agent, the server buyer agents, and the server seller agents
// (whether they participated in the auction or not)
localTask.taskBuyer.AGNT_shout_update_buyer( localTask.strTASKid, true,
intAuctionStatus, agtSERVbuyer.dblAGNTprice, random, blnOutput );
// update trading strategies: local server
agtSERVbuyer.AGNT_shout_update_buyer( strSERVid, true,
intAuctionStatus, agtSERVbuyer.dblAGNTprice, random, blnOutput );
agtSERVseller.AGNT_shout_update_seller( strSERVid, true,
intAuctionStatus, agtSERVbuyer.dblAGNTprice, random, intTimeStep,
blnOutput );
// update trading strategies: neighbouring servers
enumServers = vctSERVservers.elements();
// enumerate through all neighbouring servers
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server) enumServers.nextElement();
neighServer.agtSERVbuyer.AGNT_shout_update_buyer(
neighServer.strSERVid, true, intAuctionStatus,
agtSERVbuyer.dblAGNTprice, random, blnOutput );
neighServer.agtSERVseller.AGNT_shout_update_seller(
neighServer.strSERVid, true, intAuctionStatus,
agtSERVbuyer.dblAGNTprice, random, intTimeStep, blnOutput );
}
// ---------------------------------------------------------------
// UPDATE BANK ACCOUNTS AND ADD THE TASK TO THE SERVER
// of the server buyer agent and the server seller agent
bank( agtSERVbuyer, agtSERVbuyer.dblAGNTprice, blnOutput );
// remove task from this server
removeUDC_Task( localTask );
enumServers = vctSERVservers.elements();
// iterate through the neighbouring servers
while ( enumServers.hasMoreElements() )
{
// get the neighbouring server object
UDC_Server neighServer = (UDC_Server) enumServers.nextElement();
if ( CHOSEN_strSERVid.equals( neighServer.strSERVid ) )
{
neighServer.bank( neighServer.agtSERVseller,
agtSERVbuyer.dblAGNTprice, blnOutput );
// add task to neighbouring server
neighServer.addUDC_Task( localTask );
}
}
// ---------------------------------------------------------------
// record what the server paid for the task
localTask.dblTASKworth = agtSERVbuyer.dblAGNTprice;
blnTradeOccured = true;
```

```
}
// ---------------------------------------------------------------------
// IF NO DEAL (TASK REMAINS IN LOCAL SERVER MARKET)
else
{
// increment number of fails count (if not already set at max fails)
if ( intNoFails < MAX_FAILS )
{
intNoFails++;
}
if ( blnOutput )
{
psOUT.println( "No servers willing to sell resources (fails = "
+ intNoFails + ")" );
psOUT.println();
}
// record fact there was no deal
stats.TRADE_no_deal( intTimeStep );
// ---------------------------------------------------------------------
// UPDATE TRADING STRATEGIES
// now a trade not taken place, update the trading strategy of the
// task buyer agent, the server buyer agents, and the server seller
// agents
localTask.taskBuyer.AGNT_shout_update_buyer( localTask.strTASKid,
true, intAuctionStatus, agtSERVbuyer.dblAGNTprice, random,
blnOutput );
// LOCAL SERVER
agtSERVbuyer.AGNT_shout_update_buyer( strSERVid, true,
intAuctionStatus, agtSERVbuyer.dblAGNTprice, random,
blnOutput );
agtSERVseller.AGNT_shout_update_seller( strSERVid, true,
intAuctionStatus, agtSERVbuyer.dblAGNTprice, random,
intTimeStep, blnOutput );
// NEIGHBOURING SERVERS
enumServers = vctSERVservers.elements();
// iterate through the neighbouring servers
while ( enumServers.hasMoreElements() )
{
// get the neighbouring server object
UDC_Server neighServer = (UDC_Server) enumServers.nextElement();
neighServer.agtSERVbuyer.AGNT_shout_update_buyer(
neighServer.strSERVid, true, intAuctionStatus,
agtSERVbuyer.dblAGNTprice, random, blnOutput );
neighServer.agtSERVseller.AGNT_shout_update_seller(
neighServer.strSERVid, true, intAuctionStatus,
agtSERVbuyer.dblAGNTprice, random, intTimeStep, blnOutput );
}
}
// ---------------------------------------------------------------------
// AUCTION COMPLETED: RETURN ALL SERVERS INVOLVED IN AUCTION TO A FREE
// STATE
enumServers = vctSERVservers.elements();
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server) enumServers.nextElement();
// if neighbouring server recieved the bid for local task from this
// server
if ( localTask.strTASKid.equals(
neighServer.bidTask.strTASKid ) )
{
// RETURN SERVER INVOLVED IN AUCTION TO A FREE
// STATE
neighServer.intSERVstate = FREE;
if ( neighServer.blnEnableShifting == true )
{
// reset enable shifting flags back to false
neighServer.blnEnableShifting = false;
blnEnableOneShift = false;
}
```

```
SERVbid_server.blnStopAuction = false;
}
}
if ( intAuctionStatus == DEAL )
{
stats.STATS_calculate( intTimeStep, intAuctionStatus,
agtSERVbuyer.dblAGNTprice, TdblEqPrice, dblSurplus,
TdblCurrMaxSurp, blnOutput );
}
// this server goes back into a PENDING_ACCEPTANCE state
// in order to service the bid originally made to this
// server by another of its neighbours
intSERVstate = PENDING_ACCEPTANCE;
}
}
}
}
// ---------------------------------------------------------------------
else if ( intSERVstate == PENDING_ACCEPTANCE )
{
// A NEIGHBOURING SERVER HAS PUT OUT A BID REQUEST --
// RETURN WHETHER THIS SERVER IS WILLING OR NOT
// IF SERVER HAS ENOUGH AVAILABLE CAPACITY
if ( intSERVavail_ru >= bidTask.intTASKload )
{
// auction to be requested for participation to be closed via a
// BID
if ( blnDealType == true )
{
// get whether this server's seller agent is willing to deal
blnSERVwilling = isWilling( dblSERVbid_price, blnOutput );
// set true flag to indicate server *has* returned whether
// it is willing or not
blnSERVret_willing = true;
if ( blnSERVwilling == true )
{
// if server willing then set to wait for auction for bid
// task
intSERVstate = WAITING_AUCTION;
// (not an 'offer' auction so set offer price to 0.0)
dblSERVofferprice = 0.0;
}
else
{
// if not willing return to a free state
intSERVstate = FREE;
blnSERVwilling = false;
}
}
// auction to be requested for participation to be closed via an
// OFFER
else
{
// set true flag to indicate server *has* returned whether it
// is willing or not
blnSERVret_willing = true;
// get the offer price from the server seller agent
agtSERVseller.AGNT_calcPrice();
dblSERVofferprice = agtSERVseller.dblAGNTprice;
// if server willing then set to wait for auction for bid task
intSERVstate = WAITING_AUCTION;
}
}
// ---------------------------------------------------------------------
// IF SERVER HAS ENOUGH *POTENTIAL* CAPACITY (and running one task)
else if (( intSERVtotal_ru >= bidTask.intTASKload )
&& ( vctSERVtasks.size() == 1 ))
{
// IF TASK SHIFTING TURNED ON
if ( blnEnableShifting == true )
```

```
{
Enumeration taskEnum = vctSERVtasks.elements();
localTask = (UDC_Task) taskEnum.nextElement();
if ( blnOutput )
{
psOUT.println();
psOUT.println( "MBC_SIM: SERVER " + strSERVid +
" needs to offload (less profitable) TASK "
+ localTask.strTASKid +
" in order to partake in auction for market TASK "
+ bidTask.strTASKid );
psOUT.println( "MBC_SIM: New auction initiated" );
psOUT.println();
}
// set limit price for buyer agent on local server
// (equal to whatever price the local server originally got for
// the job)
agtSERVbuyer.dblAGNTlimit = localTask.dblTASKworth;
// calculate the price of the server buyer agent
agtSERVbuyer.AGNT_calcPrice();
// get the bid price for the server buyer agent
double dblBidPrice =
getPrice( agtSERVbuyer, strSERVid, blnOutput );
// enumerate through all neighbouring servers
Enumeration enumServers = vctSERVservers.elements();
int intNoAbleNeighbours = 0;
while ( enumServers.hasMoreElements() )
{
UDC_Server neighServer = (UDC_Server) enumServers.nextElement();
// IF NEIGHBOURING SERVER IN A FREE STATE
if ( neighServer.intSERVstate == FREE )
{
neighServer.bidTask = localTask;
neighServer.SERVbid_server = this;
neighServer.dblSERVbid_price = dblBidPrice;
neighServer.intSERVstate = PENDING_ACCEPTANCE;
//neighServer.blnEnableShifting = true;
intNoAbleNeighbours++;
}
}
// IF NO NEIGHBOURING SERVERS ABLE TO PARTAKE
if ( intNoAbleNeighbours == 0 )
{
// REMAIN IN A FREE STATE
intSERVstate = FREE;
blnSERVwilling = false;
blnSERVret_willing = true;
}
// ELSE THERE COULD BE WILLING SERVERS
else
{
// PUT SERVER INTO STATE WAITING FOR WILLING FLAGS
intSERVstate = WAITING_ACCEPTANCE;
blnSERVret_willing = false;
blnWillingWaitType = false;
}
}
// ------------------------------------------------------------------
// TASK SHIFTING TURNED OFF
else
{
intSERVstate = FREE;
blnSERVwilling = false;
blnSERVret_willing = true;
}
}
// ------------------------------------------------------------------
// TASK LOAD HIGHER THAN SERVER CAPACITY
else
{
intSERVstate = FREE;
```

```
blnSERVwilling = false;
blnSERVret_willing = true;
}
}
// ---------------------------------------------------------------
else if ( intSERVstate == WAITING_AUCTION )
{
// (REMAIN IN THIS STATE UNTIL BID SERVER COMPLETES AUCTION)
}
// ---------------------------------------------------------------
// OUTPUT CURRENT SERVER STATE
int intSERVid = Integer.parseInt( strSERVid.substring( 1 ) );
if ( blnOutput )
{
if ( intSERVstate == 0 )
{
psOUT.println( "SERVER " + strSERVid +
" in FREE STATE (Timestep = " + intTimeStep + ")" );
//System.out.println( "SERVER " + strSERVid +
//" in FREE STATE (Timestep = " + intTimeStep + ")" );
}
else if ( intSERVstate == 1 )
{
psOUT.println( "SERVER " + strSERVid +
" in PENDING_ACCEPTANCE STATE (Timestep = " + intTimeStep
+ ")" );
//System.out.println( "SERVER " + strSERVid + " in
//PENDING_ACCEPTANCE STATE (Timestep = " + intTimeStep
//+ ")" );
}
else if ( intSERVstate == 2 )
{
psOUT.println( "SERVER " + strSERVid +
" in WAITING_ACCEPTANCE STATE (Timestep = " + intTimeStep
+ ")" );
//System.out.println( "SERVER " + strSERVid +
//" in WAITING_ACCEPTANCE STATE (Timestep = " +
//intTimeStep + ")" );
}
else if ( intSERVstate == 3 )
{
psOUT.println( "SERVER " + strSERVid +
" in WAITING_AUCTION STATE (Timestep = " + intTimeStep
+ ")" );
//System.out.println( "SERVER " + strSERVid + " in
//WAITING_AUCTION STATE (Timestep = " + intTimeStep
//+ ")" );
}
}
return intTASKid;
}
// ---------------------------------------------------------------------
// ADDUDC_TASK: add and run a task on the server
public void addUDC_Task( UDC_Task task1 )
{
int TMPintSERVavail_ru = intSERVavail_ru;
// remove the necessary number of server resource units
intSERVavail_ru -= task1.intTASKload;
if ( intSERVavail_ru < 0.0 )
{
// error: not enough room on the server for the task
//psOUT.println(
//FAIL: attempting to run a task on a server without "
// + "enough capacity" );
System.out.println(
"FAIL: attempting to run a task on a server without "
+ "enough capacity" );
System.exit( 0 );
}
else
```

```
{
// set the server currently responsible for this task
task1.strTASKserv_id = strSERVid;
//psOUT.println();
//psOUT.println( "UDC_SERVER: Task " + task1.strTASKid +
//" (LOAD: " + task1.intTASKload + ", DURATION: " +
//task1.intTASKduration + ") loaded onto Server " + strSERVid
//+ " (TOT/CAP: " + intSERVtotal_ru + ", OLD/CAP: " +
//TMPintSERVavail_ru + ", NEW/CAP: " + intSERVavail_ru + ")" );
//psOUT.println();
vctSERVtasks.addElement( task1 );
}
}
// ---------------------------------------------------------------------
public void removeUDC_Task( UDC_Task task1 )
{
// add back the necessary number of server resource units
intSERVavail_ru += task1.intTASKload;
// removes the first occurance of the specified element in the
// Vector if the Vector does not contain the element, it is
// unchanged
vctSERVtasks.remove( task1 );
}
// ---------------------------------------------------------------------
public void addUDC_Server( UDC_Server server1 )
{
vctSERVservers.addElement( server1 );
}
// ---------------------------------------------------------------------
public void addUDC_Servers( UDC_Server server1, UDC_Server server2,
UDC_Server server3, UDC_Server server4, UDC_Server server5,
UDC_Server server6, UDC_Server server7, UDC_Server server8,
UDC_Server server9, UDC_Server server10 )
{
vctSERVservers.addElement( server1 );
vctSERVservers.addElement( server2 );
vctSERVservers.addElement( server3 );
vctSERVservers.addElement( server4 );
vctSERVservers.addElement( server5 );
vctSERVservers.addElement( server6 );
vctSERVservers.addElement( server7 );
vctSERVservers.addElement( server8 );
vctSERVservers.addElement( server9 );
vctSERVservers.addElement( server10 );
}
// ---------------------------------------------------------------------
public void addUDC_Servers( UDC_Network UDCnet, String strSERVid1,
String strSERVid2, String strSERVid3, String strSERVid4,
String strSERVid5, String strSERVid6, String strSERVid7,
String strSERVid8, String strSERVid9, String strSERVid10 )
{
Hashtable UDC = UDCnet.getNetwork();
// retrieve every server object in turn
for ( int intServer = 1; intServer <= UDC.size(); intServer++ )
{
// retrieve network server object
UDC_Server server = (UDC_Server) UDC.get( "S" + intServer );
if ( ( server.strSERVid.equals( strSERVid1 ) ) ||
( server.strSERVid.equals( strSERVid2 ) ) ||
( server.strSERVid.equals( strSERVid3 ) ) ||
( server.strSERVid.equals( strSERVid4 ) ) ||
( server.strSERVid.equals( strSERVid5 ) ) ||
( server.strSERVid.equals( strSERVid6 ) ) ||
( server.strSERVid.equals( strSERVid7 ) ) ||
( server.strSERVid.equals( strSERVid8 ) ) ||
( server.strSERVid.equals( strSERVid9 ) ) ||
( server.strSERVid.equals( strSERVid10 ) ) );
{
vctSERVservers.addElement( server );
```

```
}
}
}
// ---------------------------------------------------------------------
public void addUDC_Servers( Vector servers )
{
for ( int intServer = 0; intServer < servers.size(); intServer++ )
{
vctSERVservers.addElement( servers.elementAt( intServer ) );
}
}
// ---------------------------------------------------------------------
// decrement by one the duration of any tasks currently running
public void decTaskDurations( boolean blnOutput )
{
// if there are any UDC_Task objects currently running
if ( vctSERVtasks.size() > 0 )
{
// enumerate through the tasks vector
Enumeration enum = vctSERVtasks.elements();
while( enum.hasMoreElements() )
{
UDC_Task task = (UDC_Task) enum.nextElement();
task.intTASKduration -= 1;
if ( blnOutput )
{
//psOUT.println( "UDC_SERVER: SERVER " + strSERVid +
//": TASK " + task.strTASKid + " has " +
//task.intTASKduration + " seconds left to run" );
}
// if the intTASK duration of the task equals 0
if ( task.intTASKduration == 0 )
{
// add back the necessary number of RUs to the server
// (add to intAvailRUs)
intSERVavail_ru += task.intTASKload;
if ( blnOutput )
{
//psOUT.println( "UDC_SERVER: SERVER " + strSERVid
//+ ": TASK " + task.strTASKid
//+ " has completed and been removed" );
//psOUT.println();
}
// remove the UDC_Task from the UDC_Server
vctSERVtasks.remove( task );
intSERVjprsd++;
}
}
}
}
// ---------------------------------------------------------------------
public void addTaskToMarket( UDC_Task task1 )
{
vctSERVmarket.addElement( task1 );
}
// ---------------------------------------------------------------------
public void removeTaskFromMarket( UDC_Task task1 )
{
vctSERVmarket.remove( task1 );
}
// ---------------------------------------------------------------------
// return whether this server's seller agent is able to deal
public boolean isWilling( double dblPrice, boolean blnOutput )
{
// use intelligence to determine whether willing to trade at
// this price or not
agtSERVseller.AGNT_willing_trade( dblPrice );
// if agent willing
if ( agtSERVseller.blnAGNTwilling == true )
{
```

```
// get monetary reward for the deal
double dblReward = reward( agtSERVseller, dblPrice );
if ( blnOutput )
{
if ( agtSERVseller.blnAGNTtype == BUY )
{
//psOUT.println( "BUYER on server " + strSERVid +
//" willing (reserve) price = " + dblPrice
//+ " (reward = " + dblReward + ")" );
}
else
{
//psOUT.println( "SELLER on server " + strSERVid +
// " willing (reserve) price = " + dblPrice
//+ " (reward = " + dblReward + ")" );
}
}
}
//psOUT.println();
return agtSERVseller.blnAGNTwilling;
}
// ---------------------------------------------------------------------
// return whether this server's seller agent is able to deal
public boolean isBuyWilling( double dblPrice, boolean blnOutput )
{
// use intelligence to determine whether willing to trade at this
// price or not
agtSERVbuyer.AGNT_willing_trade( dblPrice );
// if agent willing
if ( agtSERVbuyer.blnAGNTwilling == true )
{
// get monetary reward for the deal
double dblReward = reward( agtSERVbuyer, dblPrice );
if ( blnOutput )
{
if ( agtSERVseller.blnAGNTtype == BUY )
{
//psOUT.println( "BUYER on server " + strSERVid +
//" willing (reserve) price = " + dblPrice +
//" (reward = " + dblReward + ")" );
}
else
{
//psOUT.println( "SELLER on server " + strSERVid +
//" willing (reserve) price = " + dblPrice
//+ " (reward = " + dblReward + ")" );
}
//psOUT.println();
}
}
return agtSERVbuyer.blnAGNTwilling;
}
// ---------------------------------------------------------------------
// given a bid or offer price, returns the monetary reward for a deal
public double reward( ZIP_Agent agent, double dblAGNTprice )
{
double dblReward = 0.0;
// if a SELLER agent
if ( agent.blnAGNTtype == SELL )
{
// reward = offer price - agent limit price
dblReward = ( dblAGNTprice - agent.dblAGNTlimit );
}
// if a BUYER agent
if ( agent.blnAGNTtype == BUY )
{
// reward = agent limit price - bid price
dblReward = ( agent.dblAGNTlimit - dblAGNTprice );
}
```

```
// ensure any negative rewards set at 0.0
if ( dblReward < 0.0 )
{
dblReward = 0.0;
}
return dblReward;
}
// ------------------------------------------------------------------
// get a (bid or offer) price for an agent
public double getPrice( ZIP_Agent agent, String strID,
boolean blnOutput )
{
double dblPrice;
// bounds on random prices
double dblRandomMin = 0.01;
double dblRandomMax = 4.0;
dblPrice = agent.dblAGNTprice;
// output bid or offer price
if ( blnOutput )
{
// get monetary reward for the deal
double dblReward = reward( agent, dblPrice );
if ( agent.blnAGNTtype == BUY )
{
//psOUT.println( "ZIP_SIM: BUYER on server " + strID +
//" BIDS AT " + dblPrice + " (reward = " + dblReward
//+ ")" );
}
else
{
//psOUT.println( "ZIP_SIM: SELLER on server " + strID
//+ " OFFERS AT " + dblPrice + " (reward = " + dblReward
//+ ")" );
}
}
return dblPrice;
}
// ------------------------------------------------------------------
// adjust bank balances of buyer or seller agent involved in a deal
public void bank( ZIP_Agent agent, double dblPrice, boolean blnOutput )
{
double dblReward;
// BUYER
if ( agent.blnAGNTtype == BUY )
{
// get monetary reward for the deal
dblReward = reward( agent, dblPrice );
// add monetary reward for the deal to the agents' bank account
agent.dblAGNTmoney += dblReward;
agent.dblAGNTactualgain += dblReward;
dblSurplus += dblReward;
if ( blnOutput )
{
//psOUT.println( "ZIP_SIM.SIM_bank: BUYER limit = "
//+ agent.dblAGNTlimit + ", reward = " + dblReward +
//", money = " + agent.dblAGNTmoney +
//", (surplus = " + dblSurplus + " )" );
}
}
// SELLER
if ( agent.blnAGNTtype == SELL )
{
// get monetary reward for the deal
dblReward = reward( agent, dblPrice );
// add monetary reward for the deal to the agents' bank account
agent.dblAGNTmoney += dblReward;
agent.dblAGNTactualgain += dblReward;
dblSurplus += dblReward;
```

```
if ( blnOutput )
{
//psOUT.println();
//psOUT.println( "ZIP_SIM.SIM_bank: SELLER limit = "
//+ agent.dblAGNTlimit + ", reward = " + dblReward +
//", money = " + agent.dblAGNTmoney +
//", (surplus = " + dblSurplus + " )" );
}
}
}
// ------------------------------------------------------------------
}
```

C.2.6 UDC_Task.java

Description: An object representing a computational task to be run on the UDC network. A ZIP agent representing the UDC_Task (on behalf of the tasks' owners) will use the task funds to allocate it resources for execution.

```
// Class: UDC_Task.java
import java.io.PrintStream;
public class UDC_Task implements ZIP_Constants
{
// id
public String strTASKid;
// duration
public int intTASKduration;
// resource requirement
public int intTASKload;
// funds to spend on resources (used by ZIP_Agents)
public double dblTASKfunds;
// funds per resource unit requirement per duration timestep
public double dblTASKpay;
// id of server currently responsible for this task
public String strTASKserv_id;
// ZIP_Agent used to purchase resources on a given server
public ZIP_Agent taskBuyer;
// what a given server paid for the task
public double dblTASKworth;
// ---------------------------------------------------------------------
public UDC_Task( String TMPstrTASKid, int TMPintTASKduration,
int TMPintTASKload, double TMPdblTASKfunds )
{
strTASKid = TMPstrTASKid;
intTASKduration = TMPintTASKduration;
intTASKload = TMPintTASKload;
dblTASKfunds = TMPdblTASKfunds;
dblTASKpay = ( dblTASKfunds / intTASKduration ) / intTASKload;
strTASKserv_id = "";
dblTASKworth = 0.0;
taskBuyer = new ZIP_Agent();
}
// ---------------------------------------------------------------------
// get a (bid or offer) price for an agent
public double getPrice( ZIP_Agent agent, String strID,
boolean blnOutput )
{
double dblPrice;
// bounds on random prices
double dblRandomMin = 0.01;
double dblRandomMax = 4.0;
dblPrice = agent.dblAGNTprice;
// output bid or offer price
if ( blnOutput )
{
// get monetary reward for the deal
double dblReward = reward( agent, dblPrice );
if ( agent.blnAGNTtype == BUY )
{
//psOUT.println( "ZIP_SIM: BUYER on task " + strID +
//" BIDS AT " + dblPrice + " (reward = "
//+ dblReward + ")" );
}
}
return dblPrice;
}
// ---------------------------------------------------------------------
// given a bid or offer price, returns the monetary reward for a deal
public double reward( ZIP_Agent agent, double dblAGNTprice )
{
```

```
double dblReward = 0.0;
// if a SELLER agent
if ( agent.blnAGNTtype == SELL )
{
// reward = offer price - agent limit price
dblReward = ( dblAGNTprice - agent.dblAGNTlimit );
}
// if a BUYER agent
if ( agent.blnAGNTtype == BUY )
{
// reward = agent limit price - bid price
dblReward = ( agent.dblAGNTlimit - dblAGNTprice );
}
// ensure any negative rewards set at 0.0
if ( dblReward < 0.0 )
{
dblReward = 0.0;
}
return dblReward;
}
// ---------------------------------------------------------------
}
```

C.2.7 UDC_Vis.java

Description: Provides a visualisation of the load on each server; the number of tasks in each servers' local market; and the quote price of each servers buyer and seller ZIP agents, at each timestep of the simulation. As such this class is not directly involved in the operation of the system.

```
// Class: UDC_Vis.java
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JFrame;
public class UDC_Vis extends JFrame
{
// visualisation parameters
private int intMaxXCells;
private int intMaxYCells;
private int intGridXScale;
private int intGridYScale;
private int intGridXOffset = 28;
private int intGridYOffset = 92;
private int intGridHeight = 400;
private int intGridWidth = 400;
private double[][] dblCellValues;
private String strField;
private int intTotalJobsPrsd;
private double dblTaskAllocPerc;
// ---------------------------------------------------------------------
public UDC_Vis( int TMPintMaxXCells, int TMPintMaxYCells,
String TMPstrField, double[][] TMPdblCellValues,
int TMPintTotalJobsPrsd, double TMPdblTaskAllocPerc )
{
intMaxXCells = TMPintMaxXCells;
intMaxYCells = TMPintMaxYCells;
strField = TMPstrField;
intTotalJobsPrsd = TMPintTotalJobsPrsd;
dblTaskAllocPerc = TMPdblTaskAllocPerc;
dblCellValues = new double[intMaxXCells][intMaxYCells];
// calculate each servers' used capacity at each timestep
for ( int intXCell = 0; intXCell < intMaxXCells; intXCell++ )
{
for ( int intYCell = 0; intYCell < intMaxYCells; intYCell++ )
{
if ( strField.equals( "LOAD" ) )
{
dblCellValues[intXCell][intYCell] =
TMPdblCellValues[intXCell][intYCell]; // * 0.5;
if ( dblCellValues[intXCell][intYCell] > 1.0 )
{
dblCellValues[intXCell][intYCell] = 1.0;
}
if ( dblCellValues[intXCell][intYCell] < 0.0 )
{
dblCellValues[intXCell][intYCell] = 0.0;
}
}
else if ( strField.equals( "WAITING TASKS" ) )
{
dblCellValues[intXCell][intYCell] =
TMPdblCellValues[intXCell][intYCell] / 10;
if ( dblCellValues[intXCell][intYCell] > 1.0 )
{
dblCellValues[intXCell][intYCell] = 1.0;
}
}
else if ( strField.equals( "OFFER PRICE" ) ||
strField.equals( "BID PRICE" ) )
{
```

```
dblCellValues[intXCell][intYCell] =
TMPdblCellValues[intXCell][intYCell] * 0.40;
if ( dblCellValues[intXCell][intYCell] > 1.0 )
{
dblCellValues[intXCell][intYCell] = 1.0;
}
}
else if ( strField.equals( "JOBS PRSD" ) ||
strField.equals( "TASKS ALLOCATED" ) )
{
dblCellValues[intXCell][intYCell] =
TMPdblCellValues[intXCell][intYCell] / 5;
if ( dblCellValues[intXCell][intYCell] > 1.0 )
{
dblCellValues[intXCell][intYCell] = 1.0;
}
}
else if ( strField.equals( "PERCENTAGE ALLOCATED" ) )
{
dblCellValues[intXCell][intYCell] =
TMPdblCellValues[intXCell][intYCell] / 25;
if ( dblCellValues[intXCell][intYCell] > 1.0 )
{
dblCellValues[intXCell][intYCell] = 1.0;
}
}
else
{
System.out.println(
"FAIL: incorrect UDC visualisation field given" );
System.exit( 0 );
}
}
}
setSize( 550, 550 );
      setTitle( "MBC_Sim" );
      setResizable( false );
}
// ---------------------------------------------------------------------
// VIS_SCALE_GRID: scale 'grid' according to number of servers (x axis)
// and number of timesteps (y axis)
public void VIS_scale_grid()
{
intGridXScale = ( ( intGridWidth - 2 ) / ( intMaxXCells + 1 ) );
intGridYScale = ( ( intGridHeight - 2 ) / ( intMaxYCells + 1 ) );
// sets cells as squares rather than rectangles
//if ( intGridXScale > intGridYScale )
//{
// intGridXScale = intGridYScale;
//}
//else
//{
// intGridYScale = intGridXScale;
//}
}
// ---------------------------------------------------------------------
// PAINT: draws cell grid
public void paint( Graphics g )
{
g.drawString( "SERVER " + strField + " VISUALISATION", 30, 48 );
if ( strField.equals( "JOBS PRSD" ) )
{
g.drawString( "TOTAL NO. OF TASKS PROCESSED BY ALL SERVERS: "
+ intTotalJobsPrsd, 30, 67 );
}
else if ( strField.equals( "TASKS ALLOCATED" ) )
{
g.drawString( "TOTAL NO. OF TASKS ALLOCATED BY ALL SERVERS: "
+ intTotalJobsPrsd, 30, 67 );
}
else
```

```
{
g.drawString( "Servers: " + intMaxXCells +
" (X-AXIS); PERCENTAGE OF TASKS ALLOCATED: "
+ dblTaskAllocPerc + "%", 30, 67 );
g.drawString( "Timesteps: " + intMaxYCells +
" (Y-AXIS)", 30, 86 );
}
VIS_scale_grid();
// draw grid
for ( int intXCell = 0; intXCell < intMaxXCells; intXCell++ )
{
for ( int intYCell = 0; intYCell < intMaxYCells; intYCell++ )
{
if ( strField.equals( "OFFER PRICE" ) ||
strField.equals( "LOAD" ) ||
strField.equals( "JOBS PRSD" ) ||
strField.equals( "BID PRICE" ) ||
strField.equals( "TASKS ALLOCATED" ) ||
strField.equals( "PERCENTAGE ALLOCATED" ) )
{
int intLH = (int)
( dblCellValues[intXCell][intYCell] * 255 );
int intHL = (int) ( 255 -
( dblCellValues[intXCell][intYCell] * 255 ));
if ( intLH > 255 )
{
intLH = 255;
}
if ( intHL > 255 )
{
intHL = 255;
}
if ( intLH < 0 )
{
intLH = 0;
}
if ( intHL < 0 )
{
intHL = 0;
}
if ( dblCellValues[intXCell][intYCell] < 0.5 )
{
g.setColor( new Color( ( intLH * 2 ), 255, 0 ) );
}
else
{
g.setColor( new Color( 255, (intHL * 2 ), 0 ) );
}
}
else
{
int intShade = (int) ( 255 -
( dblCellValues[intXCell][intYCell] * 255 ));
g.setColor( new Color( 255, intShade, intShade ) );
if ( intShade > 255 )
{
intShade = 255;
}
}
g.fillRect( (intXCell * intGridXScale) + intGridXOffset
+ intGridXScale + 1, (intYCell * intGridYScale) +
intGridYOffset + intGridYScale + 1, intGridXScale,
intGridYScale );
}
}
}
// ---------------------------------------------------------------------
}
```

Bibliography

[1] **Friedrich, R.**, *Utility Computing on a Planetary Scale*, mpulse Magazine, Cooltown Publications, January 2002, http://www.cooltown.com/mpulse/0102-thinker.asp.

[2] **Waldspurger, C. A., Hogg, T., Huberman, B. A., Kephart, J. O.** and **Stornetta, S.**, *Spawn: A Distributed Computational Economy*, IEEE Transactions on Software Engineering, Vol. 18, No. 2, pp. 103 - 117, 1992.

[3] **Cliff, D.**, *Minimal-Intelligence Agents for Bargaining Behaviors in Market-Based Environments*, Tech. Rep. HP-1997-91, Hewlett Packard Laboratories, Bristol, UK, 1997.

[4] **Das, R., Hanson, J. E., Kephart, J. O.** and **Tesauro, G.**, *Agent-Human Interactions in the Continuous Double Auction*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Seattle, August 2001, 2001.

[5] **Andrzejak, A., Graupner, S., Kotov, V.** and **Trinks, H.**, *Self-Organizing Control in Planetary-Scale Computing*, IEEE International Symposium on Cluster Computing and the Grid (CCGrid), 2nd Workshop on Agent-based Cluster and Grid Computing (ACGC), 2002.

[6] **Andrzejak, A., Graupner, S., Kotov, V.** and **Trinks, H.**, *Control Architecture for Service Grids in a Federation of Utility Data Centres*, Hewlett Packard Laboratories, Palo Alto, USA (forthcoming), 2002.

[7] **D.Begg, S.Fischer** and **Dornbusch, R.**, Economics (fourth edition), McGraw-Hill, London, 1994.

[8] **Clearwater, S.**, Market-Based Control: A Paradigm for Distributed Resource Allocation, World Scientific, Singapore, 1996.

[9] **Cliff, D.** and **Bruten, J.**, *Simple Bargaining Agents for Decentralized Market-Based Control*, Tech. Rep. HP-98-17, Hewlett Packard Laboratories, Bristol, UK, 1998.

[10] **Wolski, R.**, **Plank, J.** and **Bryan, T.**, *Analyzing Market-based Resource Allocation Strategies for the Computational Grid*, Tech. Rep. CS-00-453, University of Tennessee, Knoxville, USA, 2000.

[11] **Smale, S.**, *Convergent process of price adjustment and global newton methods*, Contributions to Economic Analysis, 105: pp. 191-205, 1977.

[12] **M. A. Gibney, N. R. J.**, **Vriend, N. J.** and **Griffiths, J. M.**, *Market-Based Call Routing in Telecommunications Networks using Adaptive Pricing and Real Bidding*, 3rd International Workshop on Multi-Agent Systems and Telecommunications (IATA-99), pp. 50-65, 1999.

[13] **Smith, R.**, *The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver*, IEEE Transactions on Computers, 29:12, pp. 1104 - 1113, 1980.

[14] **Gode, D.** and **Sunder, S.**, *Allocative efficiency of markets with zero-intelligence traders: Market as a partial substitute for individual rationality*, Journal of Political Economy, 101(1): pp. 119-137, 1993.

[15] **Smith, V. L.**, *An experimental study of competitive market behavior*, Journal of Political Economy, 70: pp. 111-137, 1962.

[16] **Cliff, D.**, *Evolutionary Optimization of Parameter Sets for Adaptive Software-Agent Traders in Continuous Double Auction Markets*, Tech. Rep. HP-2001-99, Hewlett Packard Laboratories, Bristol, UK, 2001.

[17] **Montfort, G. V.**, *Economic Agents for Controlling Complex Systems*, Master's thesis, Delft University of Technology Faculty of Technical Mathematics and Informatics, Unpublished, September 1997.

[18] **Cliff, D.**, *Evolution of Market Mechanism Through a Continuous Space of Auction-Types*, Tech. Rep. HP-2001-326, Hewlett Packard Laboratories, Bristol, UK, 2001.

[19] **Cliff, D.**, *Evolution of Market Mechanism Through a Continuous Space of Auction-Types II: Two-sided auction mechanisms evolve in response to market shocks*, Tech. Rep. HP-2002-128, Hewlett Packard Laboratories, Bristol, UK, 2002.

[20] **Hogg, T.** and **Huberman, B.**, *Dynamics of Large Autonomous Computational Systems*, Tech. Rep. HP-2002-77, Hewlett Packard Laboratories, Palo Alto, USA, 2002.

[21] **Hillis, D.**, *Co-Evolving Parasites Improve Simulated Evolution as an Optimization Procedure*, Physica D, 42: pp. 228-234, 1990.

[22] **Santos, C.**, **Zhu, X.** and **Crowder, H.**, *A Mathematical Optimization Approach for Resource Allocation in Large Scale Data Centers*, Tech. Rep. HP-2002-64, Hewlett Packard Laboratories, Palo Alto, USA, 2002.